Testauksen integrointi ohjelmistokehitysprosessiin

Ville Kukkola

Pro-gradu tutkielma
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen osasto

Helsinki, 28. huhtikuuta 2020

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen	Tietojenkäsittelytieteen osasto

Tekijä — Författare — Author
Ville Kukkola

Työn nimi — Arbetets titel — Title

Testauksen integrointi ohjelmistokehitysprosessiin

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Pro-gradu tutkielma	28. huhtikuuta 2020	50

Tiivistelmä — Referat — Abstract

Ohjelmistotestaus on oleellista ohjelmiston toiminnan varmistamisessa. Testauksessa varmistetaan, että tuotettu ohjelmisto vastaa sille asetettuja määrityksiä. Testaus on kuitenkin aikaa ja resursseja kuluttavaa ja ohjelmisto tulee testata jokaisen julkaisun yhteydessä. Ohjelmistokehitys on poikkeuksetta iteratiivista, jolloin ohjelmistosta julkaistaan useita versioita sen kehityksen aikana. Testauksen työmäärää voidaan vähentää automaatiotestauksella.

Automaatiotestauksessa käytetään erilaisia testauskehyksiä, jotka toteuttavat yksittäisten tilanteiden testauksen ja testien automaattisen suorituksen. Kohdeyrityksellä on useita Java-ohjelmia, joiden kehityksessä ei ole huomioitu automaatiotestausta. Tarkastelemme ohjelmistokehitysalalla yleisesti käytössä olevia testauskehyksiä ja arvioimme niiden soveltuvuutta kohdeyrityksen ohjelmistokehitysprosessiin.

Tarkastelemme yleisesti käytössä olevia yksikkötestaus-, mockaus- ja käyttöliittymätestauskehyksiä. Tarkasteltavia testauskehyksiä haetaan alan ei tieteellisistä julkaisuista, jolloin voidaan varmistaa, että tarkastellut kehykset ovat ohjelmistokehitysalalla suositeltuja ja yleisesti käytössä. Testauskehyksiä arvioidaan kohdeyrityksen prosessin näkökulmasta.

Tutkimuksen pohjalta automaatiotestauksen integrointi ohjelmistokehitysprosessiin on oleellista. Vähintään yksikkötestauksen käyttöönotto on vaadittua ja tarpeen mukaan mockauksen ja käyttöliittymätestauksen käyttöönottoa voidaan harkita. Kaikki tarkastellut testauskehykset soveltuvat kohdeyrityksen tarpeisiin, joten niistä voidaan valita haluttu kombinaatio.

ACM Computing Classification System (CCS):
Software and its engineering → Software creation and management → Software verification and validation → Software defect analysis → Software testing and debugging

Avainsanat — Nyckelord — Keywords
automaatiotestaus, ohjelmistokehitysprosessi, testauksen integrointi

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Sisältö

1 Johdanto

Testaus on olennainen osa ohjelmistokehitystä. Testauksessa tarkastetaan, että kehitetty ohjelmisto vastaa määrittelyjä, joiden mukaan ohjelmistoa kehitetään. Testauksen tarkoituksena on varmistaa, että ohjelmaan tai varusohjelmistoihin tehdyt muutokset eivät aiheuta odottamattomia seurauksia ohjelmassa. On oleellista, että ohjelmisto testataan jokaisen uuden version julkaisun yhteydessä.

Testaus voidaan suorittaa joko manuaalisesti tai automaattisesti. Manuaalisessa testauksessa ohjelmaa suoritetaan tietyin parametrein tai ohjelmassa tehdään tietyt toiminnot. Automaattisessa testauksessa, tai automaatiotesteissä, kirjoitetaan ohjelmalle testejä, jotka suoritetaan ohjelman kääntämisen yhteydessä. Automaatiotesteissä voidaan myös tehdä erilaisia testipaketteja, jolloin pienten muutosten yhteydessä voidaan suorittaa pienempi osa testejä ja ennen muutosten integrointia voidaan suorittaa kaikki testit, jolloin varmistetaan, että muutokset eivät ole aiheuttaneet odottamattomia seurauksia.

Automaattisen testauksen etuna on, että manuaalinen testaus on työlästä, erityisesti silloin, kun testattavia tilanteita on paljon tai uusia versioita ohjelmistosta julkaistaan usein. Testauksessa tulisi myös varmistaa, että aiemmin toimivaksi todetut osat ohjelmaa ovat edelleen toimivia. Manuaalisella testauksella tämä tuottaa mahdottoman määrän työtä testaajalle, erityisesti silloin kun kehittäjä itse hoitaa testaamisen. Automaattisessa testauksessa testit pitää kirjoittaa kerran, jonka jälkeen olemassa oleville ominaisuuksille suoritetaan testit ilman mitään lisätyötä.

Testaaminen voidaan yleisesti jakaa eri tasoihin. Yksikkötesteillä testataan yksittäisen koodisegmentin, esimerkiksi funktion, toiminta. Integraatiotesteillä testataan suurempien ohjelman osien, esimerkiksi luokkien, toiminta. Järjestelmätesteillä testataan koko ohjelman toiminta ja varmistetaan, että se vastaa vaatimuksia ja määritelmiä.

Yksikkötesteissä pyritään varmistamaan, että yksittäinen komponentti toimii vaatimustensa mukaisesti. Haasteena on, että jo kohtuullisen yksinkertaisissa funktioissa erilaisia suoritustapoja ja rajatapauksia voi tulla useita kappaleita. Tämä johtaa siihen, että jo yksittäiselle komponentille voidaan joutua tekemään lukuisia testejä, joka johtaa komponenttien määrän lisääntyessä testien kokonaismäärän valtavaan kasvuun. Tämän takia on oleellista, että testejä tehtäessä valitaan ohjelmiston kannalta oleelliset testitapaukset.

Integraatiotestauksessa varmistetaan sekä ohjelman sisäisten rajapintojen toiminta että ulkoisten rajapintojen toiminta ohjelman sisäisen logiikan kanssa. Sisäisten rajapintojen tapauksessa varmistetaan, että esimerkiksi kaksi luokkaa tai vastaavaa komponenttia toimivat yhdessä määritelmän mukaan. Jos molemmat luokat läpäisevät yksikkötestinsä, mutta luokkien yhteistoiminnassa on vika, on kyseessä integraatiotason virhe. Ulkoisten rajapintojen tapauksessa varmistetaan, että ohjelma käsittelee ulkoiseen

rajapintaan lähetettyä tai sieltä saatua dataa oikein. Integraatiotestaus voi jakautua usealle tasolle. Useampi luokka saattaa muodostaa alijärjestelmän, jolla on rajapinta toisen alijärjestelmän tai ulkoisen rajapinnan kanssa.

Järjestelmätestauksessa testataan ohjelmisto tuotantoympäristöä vastaavassa testausympäristössä. Testausympäristössä tulee olla kaikki tarpeelliset ohjelmiston osat toteutettuna ja siellä varmistetaan, että koko ohjelmisto toimii loppukäyttäjän määrittelemällä ja olettamalla tavalla. Järjestelmätestauksen osana on myös dokumentaation ja ei-ohjelmallisten resurssien testaus.

Testimäärää helpottaa myös suoritettavien testien rajaus. Esimerkiksi jos komponenttiin tai sen riippuvuuksiin ei ole tehty muutoksia, ei ole tarpeellista suorittaa testejä komponentille. Rajattu testien suoritus on hyödyllistä kehittäjälle. Kehittäjä voi tehdä muutoksia komponenttiin, korjata bugeja tai lisätä uusia ominaisuuksia, suorittaa rajatun määrän testejä ja tehdä mahdollisesti lisää muutoksia. Kun kehittäjä on valmis, suoritetaan kaikki testit ennen kuin uusi ohjelmaversio siirtyy tuotantoon. Täyden testimäärän suoritus voidaan siirtää pois kehittäjältä esimerkiksi jatkuvan integraation (Continuous Integration) palvelimelle, joka hoitaa testaamisen ja testien läpäistyä siirtää ohjelmaan eteenpäin julkaisuputkessa. Tässä etuna on, että kehittäjän resursseja ei kulu mahdollisesti suuren testimäärän suorittamisen odotteluun.

Tässä tutkielmassa selvitetään eri tapoja integroida testaus ohjelmistokehitysprosessiin ja erityisesti testauksen integrointia olemassa olevaan ohjelmistokehitysprojektiin. Tutkimuksessa tehdään erään Suomalaisen yrityksen Java-ohjelmistokehitystä varten (jatkossa yritys tai kohdeyritys). Yrityksellä on käytössä useita jaettuun koodipohjaan perustuvia Java-ohjelmia, joita luotaessa ja kehitettäessä ei ole huomioitu testausta. Ohjelmistojen jatkokehityksen kannalta automatisoidun testauksen integrointi ohjelmistokehitysprosessiin on oleellista.

Kohdeyrityksen ohjelmistokehitysprosessi on iteratiivinen prosessi, joka ei pohjaudu mihinkään nimettyyn prosessiin. Ohjelmistoja kehitetään projektiluontoisesti, projektissa todetaan tarve jollekin ohjelmistolle tai lisäominaisuuksille olemassa olevaan ohjelmistoon. Projektin aikana ohjelmistosta tuotetaan iteratiivisesti versioita, kunnes projektin ja asiakkaan tarpeet täyttävä ohjelmisto on valmis. Ohjelmistokehitysprojektin jälkeen seuraa usein ylläpitoprojekti, jossa asiakkaalle tarjotaan teknisen tuen lisäksi mahdollisia bugikorjauksia ja ohjelmiston päivitettyjä versioita. Ylläpitoprojektien lisäksi asiakkaan tarpeisiin voidaan aloittaa uusi projekti, jossa ohjelmistoon päivitetään uusia ominaisuuksia.

kohdeyrityksen prosessissa ohjelmistosta tehdään useita julkaisuja ja ohjelmistoon tehdään julkaisujen välillä mahdollisesti merkittäviä muutoksia. Näin ollen on oleellista, että ohjelmiston toiminta varmistetaan julkaisujen yhteydessä.

Ohjelmistot on toteutettu Java versio 8:lla ja ne on kehitetty Maven-

ympäristöllä. Ohjelmistojen kokoluokka on noin 50 000–100 000 riviä koodia. Ohjelmistojen koon takia jälkikäteen manuaalisesti testien kirjoittaminen ei ole järkevää. Jatkokehityksessä manuaalinen testien kirjoittaminen uusille ominaisuuksille on kuitenkin realistinen mahdollisuus.

Luvussa 2 käydään läpi tarvittavat taustatiedot ja oleellinen termistö. Luvussa 3 kuvataan tutkimuksen toteutus ja kriteeristö, jolla arviointi ja vertailu suoritetaan. Luvussa 4 listataan tutkimuksessa löydetyt raakatulokset. Luvussa 5 vastataan esitettyihin tutkimuskysymyksiin tarkasteltujen tulosten pohjalta, arvioidaan tutkimusjärjestelyjä ja tutkielmaan liittyviä muita tutkimuksia. Luvussa 6 käydään tutkielman pääpiirteet läpi yhteenvetona.

2 Johdatus testiautomaatioon

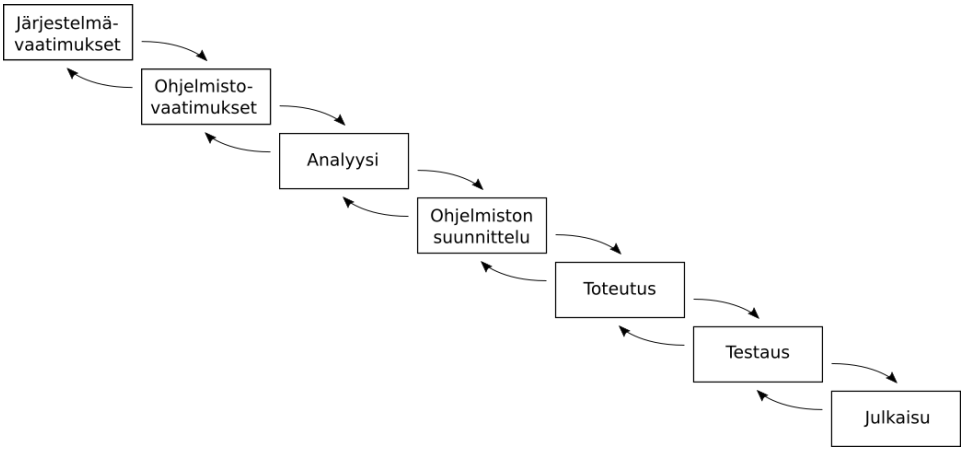
Tässä luvussa käydään läpi tutkielmassa käsiteltävien aiheiden taustatietoja ja terminologiaa. Aliluvussa 2.1 käydään läpi ohjelmistokehitysprosessin vaiheita ja niiden sisältöä. Aliluvussa 2.2 käydään läpi ohjelmistotestauksen perusteet ja motivaatio. Aliluvussa 2.3 käydään läpi tasot, joihin ohjelmistotestaus jakautuu. Aliluvussa 2.4 käsitellään mockauksen merkitys ja perusteet. Aliluvussa 2.5 käsitellään testauksen automatisoinnin tarkoitus ja yleisimpiä tekniikoita. Aliluvussa 2.6 käsitellään testimäärän rajoittamista.

2.1 Ohjelmistokehitysprosessin vaiheet

Ohjelmistokehitysprosessi voidaan yleisesti jakaa seuraaviin vaiheisiin: määrittely, suunnittelu, toteutus, testaus, hyväksyntä sekä ylläpito ja jatkokehitys [40]. Riippuen käytetystä prosessista näiden vaiheiden määrä ja järjestys voi vaihdella. Esimerkiksi ketterissä prosesseissa [24] voidaan toistaa suunnittelu-, toteutus-, testaus- ja hyväksyntävaiheita jokaisessa iteraatiossa.

Kuva 1 kuvastaa perinteisempää vesiputousmallia, jossa eri vaiheet on kuvattu niiden loogisessa järjestyksessä. Vaiheiden välillä olevaa molempiin suuntiin tapahtuvaa interaktiota on kuvattu nuolilla. Vesiputousmalli perustuu ajatukseen, että ohjelmistokehityksessä voitaisiin käyttää samanlaista prosessia kuin perinteisillä tuotantoaloilla. Tässä prosessissa jokainen vaihe suoritetaan valmiiksi ennen seuraavan vaiheen aloittamista. Esimerkiksi testaus aloitetaan vasta kun koko ohjelmisto on toteutettu. Käytännössä tällainen prosessi on todettu toimimattomaksi ohjelmistotuotannossa ja ohjelmistokehitysalalla on siirrytty käytännössä erilaisiin iteratiivisiin prosesseihin.

Iteratiivisessa prosessissa toistetaan jotakin prosessin osaa, jota kutsutaan iteratiiviseksi osaksi. Iteratiivisessa prosessissa tarkoituksena on tuottaa jokin osa tai versio ohjelmistosta jokaisessa iteraatiossa siten, että iteraatioiden edetessä ohjelmisto lähestyy valmistumista. Kuva 2 antaa esimerkin eri vaiheiden sijoittumisesta ja toistumisesta iteratiivisessa mallissa.

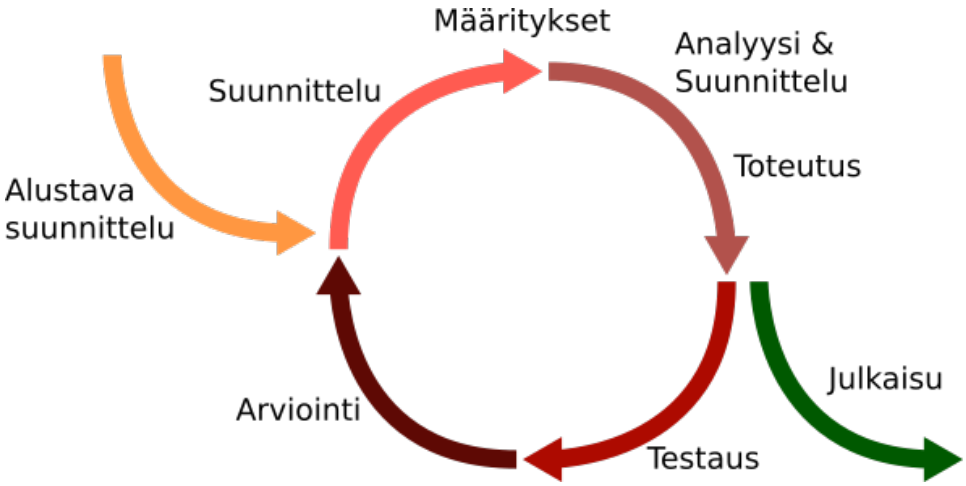


Kuva 1: Perinteinen vesiputousmalli, Winston W Royce [40] Figure 3.

Prosessi alkaa alustavasta suunnittelusta (**Initial Planning**, jossa määritetään ohjelmiston tarve ja alustavat vaatimukset, joiden pohjalta ohjelmiston kehitys alkaa. Tämän jälkeen alkaa iteratiivinen osa jota toistetaan, kunnes ohjelmisto on valmis tai prosessi päätetään muista syistä lopettaa. Iteraatio alkaa suunnitteluvaiheella (**Planning**), jossa luodaan määritykset (**Requirements**) iteraatiossa tuotettavalle toiminnallisuudelle. Vaatimusten pohjalta alkaa analyysi ja suunnitteluvaihe (**Analysis & Design**), jossa vaatimukset analysoidaan ja niiden pohjalta luodaan toteutus (**Implementation**). Toteutettu ohjelmisto julkaistaan (**Deployment**). Julkaistu ohjelmisto testataan (**Testing**). Testauksessa varmistetaan, että ohjelmiston uudet ja vanhat ominaisuudet toimivat määritysten mukaisesti. On oleellista varmistaa, että uudet ominaisuudet eivät ole aiheuttaneet virheitä vanhoihin toiminnallisuuksiin. Arviointivaiheessa (**Evaluation**) tarkastellaan sekä projektin että prosessin tilaa. Projektin arvioinnissa tarkastellaan tarvetta projektin jatkamiselle ja valmistellaan seuraavan iteraation suunnitteluvaihetta. Prosessin arvioinnissa tarkastellaan prosessin toimivuutta ja mahdollisia toimintatapojen muutoksia tulevien iteraatioiden ja projektien tehostamiseksi.

Ketterän ohjelmistokehityksen tavoitteena on julkaista mahdollisimman usein päivitetty versio ohjelmasta, uusi versio voidaan julkaista kehityshaaraan (Development Branch/Devbranch) erillisenä versiona vakaasta julkaisusta (Stable Release), jolloin testaajille ja käyttäjille on tarjolla sekä todennetusti toimiva versio että uuden toiminnallisuuden sisältävä versio.

Eri prosesseissa näitä vaiheita voidaan yhdistää tai jakaa pienempiin osiin. Esimerkiksi testivetoisessa kehityksessä (**Test Driven Development**) [23] testaus ei ole itsenäinen vaihe, vaan osa suunnittelu- ja toteutusvaiheita. Käytännössä ohjelmistokehityksessä nämä vaiheet eivät seuraa hierarkkisesti toisiaan, vaan niillä on keskinäistä vuorovaikutusta, esimerkiksi jos



Kuva 2: Iteratiivinen ohjelmistokehitysmalli, A B M Moniruzzaman ja Syed Hossain [35] Figure 3.

testauksessa havaitaan puutteita jossakin toiminnossa, palataan takaisin toteutusvaiheeseen, jossa nämä puutteet korjataan.

Määrittelyvaiheessa asetetaan vaatimukset ohjelmiston osille ja toiminnallisuuksille. Määrittelyvaiheessa ei vielä oteta kantaa käytettyihin tekniikoihin tai muihin toteutuksen yksityiskohtiin. Vaatimuksia voidaan määritellä esimerkiksi käyttötapauksina (**user story**), joissa määritetään, kuka tilanteessa toimii, mitä tilanteessa tehdään ja mikä on oletettu lopputulos.

Suunnitteluvaiheessa valitaan toteutuksessa käytettävät tekniikat, kuten ohjelmointikieli ja mahdolliset ohjelmistokehykset (**framework**). Suunnitteluvaiheessa määritellään ohjelmiston rakenne ja arkkitehtuuri sekä mahdolliset rajapinnat ja luokkarakenteet.

Toteutusvaihe, eli ohjelmointivaihe sisältää ohjelman ja muiden resurssien, kuten tietokantojen, luomisen.

Testausvaiheessa varmistetaan, että toteutettu ohjelma toimii suunnitellulla tavalla. Testausvaiheessa voidaan suorittaa sekä automaatiotestejä että manuaalisia testejä. Prosessista ja organisaatiosta riippuen testaaja voi olla sama tai eri henkilö kuin itse ohjelmoija.

Hyväksyntävaiheessa tarkastetaan, että ohjelmisto toteuttaa sille määritysvaiheessa asetetut vaatimukset ja se toteuttaa kaikki vaaditut toiminnallisuudet.

Ylläpitovaihe kattaa kaikki toimenpiteet, jotka ohjelmiston kehittäjän tulee hoitaa ohjelmiston valmistumisen ja julkaisun jälkeen. Näihin kuuluvat muun muassa palvelinten ylläpito, bugikorjaukset ja tekninen tuki. Jatkokehitysvaihe voi muistuttaa pientä ohjelmistokehitysprosessia, jossa kokonaisen ohjelmiston sijaan lisätään tai muokataan yhtä tai useampaa ohjelmiston osaa.

2.2 Ohjelmistotestaus

Verifiointi ja validointi ovat laadunvarmistusprosesseja. Verifioinnissa varmistetaan, että järjestelmä vastaa sille asetettuja vaatimuksia. Validoinnissa varmistetaan, että järjestelmä toteuttaa loppukäyttäjän tarpeen [43]. Verifiointiin ja validointiin muodostuu monenlaisia työkaluja ja menetelmiä, joihin testaus kuuluu [38]. Verifiointi ja validointi eivät kuulu mihinkään yksittäiseen prosessin vaiheeseen, vaan niitä tehdään menetelmästä riippuen jatkuvasti prosessin aikana.

Ohjelmistotestauksen tarkoitus on varmentaa, että kehitettävä ohjelmisto toimii oikein. Testauksen tavoitteena on kasvattaa ohjelmiston laatua havaitsemalla ja poistamalla ohjelmistossa olevia vikoja [42]. On oleellista huomata, että yhden vian korjaaminen voi aiheuttaa uusia vikoja, joten ohjelmiston testaaminen ei ole kertaluontoinen toimenpide.

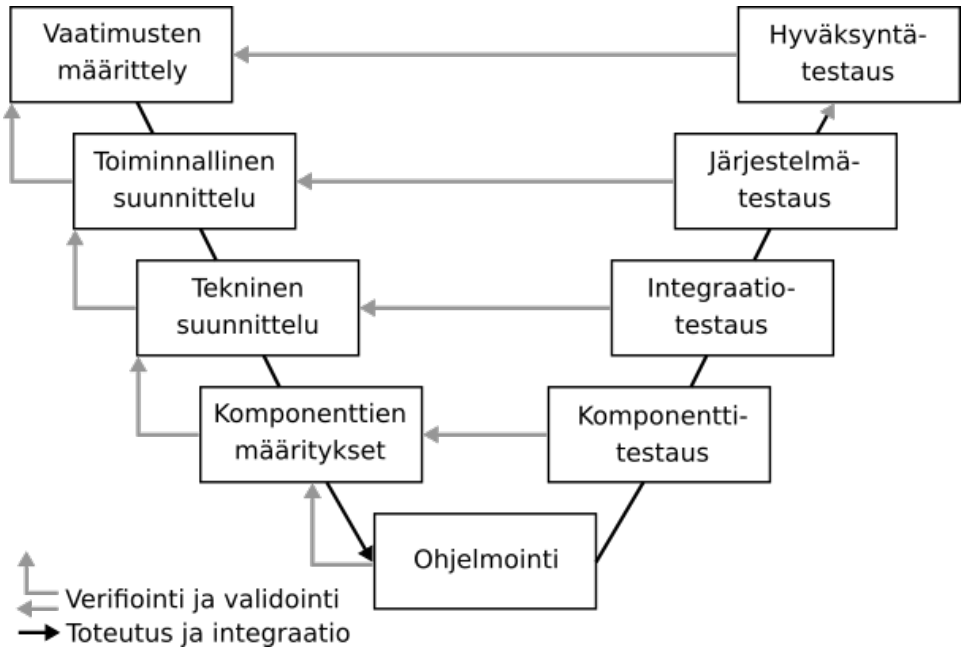
2.3 Testaustasot

Ohjelmistotestaus voidaan jakaa erilaisiin testaustasoihin testauksen tavoitteiden ja rajauksen perusteella sekä sen mukaan, mihin ohjelmistoprosessin vaiheeseen testit sijoittuvat.

Ohjelmistotestauksen v-mallilla [28] voidaan havainnollistaa eri testaustasojen suhdetta ohjelmiston määrittelyihin. Mallissa määritykset ja testit ovat samalla korkeudella, jos testien suoritus vastaa määrityksen tarkistamista. Kuvassa 3 kuvataan ohjelmistokehityksen v-malli ja siinä esiintyvät määrittely- ja testaustasot. V-mallissa eri verifikaation ja validaation vaiheet esitetään v-kirjaimen oikeassa haarassa ja eri määrittelytasot v-kirjaimen vasemmassa haarassa. Mallin tarkastelu aloitetaan vasemmasta yläkulmasta ja liikuttaessa kohti keskiosaa määrittelyt siirtyvät tarkempiin ja yksityiskohtaisempiin tasoihin. Liikuttaessa testauksen puolelle aloitetaan komponenttitason testeistä, jotka vastaavat komponenttitason määrityksiä. Lopulta päästään hyväksymistesteihin, jotka vastaavat ohjelmiston vaatimuksia.

Iteratiivisessa ohjelmistokehityksessä v-malli kuvastaa yhden iteraation määrityksiä ja testejä. Jokaisessa iteraatiossa toteutettavalle toiminnallisuudelle tulee iteraation suunnitteluvaiheessa luoda määritykset, joille luodaan testit, jotka suoritetaan iteraation testausvaiheessa. Tarpeen mukaan vanhan toiminnallisuuden testit suoritetaan regressiotesteinä. Iteratiivisessa prosessissa v-malli kuvastaa enemmän testitasojen ja suunnittelutasojen vastaavuutta kuin suoraviivaista prosessia.

Komponenttitason testaus toteutetaan yksikkötesteinä. IEEE:n ylläpitämän terminologiastandardin mukaan yksikkötestaus on: ”testing of individual routines and modules by the developer or an independent tester.” [21] Yksikkötesteissä keskitytään testaamaan yksittäisten komponenttien, kuten metodien ja olioiden toimintaa. Yksittäisessä yksikkötestissä pyritään testaamaan mahdollisimman pieniä yksittäisiä kokonaisuuksia, yksittäinen metodi



Kuva 3: Ohjelmistotestauksen v-malli, Spillner et al [42] sivu 40.

voidaan suorittaa yksikkötestissä useilla syötteillä ja on usein tarpeellista varmistaa kaikki mahdolliset rajatapaukset, jotka voidaan kohdata ohjelman suorituksen aikana toimivat odotetulla tavalla.

Integraatiotestauksessa testataan erillisten komponenttien yhteistoiminta [31]. Integraatiotesteissä testataan rajapintojen ja muiden yhdessä toimivien luokkien ja moduulien keskinäinen toiminta. Integraatiotestaus olettaa, että komponenttien sisäisessä toiminnassa ei virheitä, vaan kaikki epäonnistuneet testit johtuvat komponenttien virheellisestä yhteistoiminnasta. Tämän takia integraatiotestaus tulee suorittaa komponenttitestauksen jälkeen.

Järjestelmätestauksessa varmennetaan, että koko ohjelmisto toimii sille laadittujen määritysten mukaisesti [37]. Järjestelmätestauksessa testataan ohjelmallisten elementtien lisäksi fyysiset elementit, kuten palvelimet ja muu infrastruktuuri. Järjestelmätestauksen tavoitteena on testata ohjelmistojärjestelmä mahdollisimman samanlaisessa ympäristössä, kuin sitä tullaan käyttämään. Järjestelmätestauksessa ei olla kiinnostuneita yksittäisten komponenttien tai rajapintojen toiminnasta, nämä on jo testattu komponentti- ja integraatiotesteissä, vaan järjestelmätesteissä keskitytään ohjelmiston määrityksiin.

Hyväksyntätestauksessa ohjelmistojärjestelmän tilaaja tai käyttäjä varmistaa, että ohjelmisto toimii sen määritelmien mukaisesti [34]. Hyväksyntätestaus on tavoitteiltaan hyvin samanlainen kuin järjestelmätestaus, suurimpana erona on testin suorittaja.

Regressiotestauksessa pyritään varmistamaan ohjelmiston toiminta tilanteissa, joissa sen rakenteeseen tai komponentteihin on tehty merkittäviä muutoksia [30]. Regressiotestaus eroaa aiemmin esitetyistä testausmuodoista siinä, että se ei sijoitu ohjelmistokehitysprosessin testausvaiheeseen. Regressiotestaus sijoittuu perinteisemmissä malleissa ylläpitovaiheeseen, mutta sitä voidaan hyödyntää iteratiivisissa maleissa varmistamaan, että uudet ominaisuudet tai muutokset ohjelmistossa eivät aiheuta ongelmia vanhemmille ominaisuuksille ja toiminnoille.

Käyttöliittymätestauksessa (UI- tai GUI-testaus) varmistetaan ohjelman käyttöliittymän toiminta. Käyttöliittymätestausta voidaan pitää osana järjestelmätestausta, koska siinä varmistetaan, että ohjelman käyttäjälle esitettävä osa toimii vaaditulla tavalla [33]. Käyttöliittymätestauksessa varmistetaan, että ohjelman käyttöliittymä vastaa ohjelman sisäistä tilaa ja käyttäjän toimintoihin reagoidaan oikealla tavalla.

2.4 Mockaus

Mockaus (**mocking**) on tekniikka, jossa konkreettisen toteutuksen sijaan testauksessa käytetään simuloituja olioita [32]. Näitä simuloituja olioita kutsutaan mock-olioiksi. Yksikkötestauksessa mock-olioiden etu on, että testi on riippuvainen ainoastaan testattavasta komponentista ja muut riippuvuudet voidaan välttää käyttämällä niiden sijaan mock-olioita. Mock-oliot toteuttavat ainoastaan testauksessa tarvittavat toiminnot ja muutokset testattavan komponentin riippuvuuksissa eivät muuta mock-olioiden toimintaa.

Esimerkiksi jos testataan luokkaa `Kauppa`, muutokset muissa luokissa eivät saa vaikuttaa testien onnistumiseen. Testeissä luokat `Asiakas` ja `Kassa` voidaan toteuttaa mock-olioina. `Kauppa`-luokan testeihin ei vaikuta miten `Asiakas`-luokka on toteuttanut tuotteen lisäämisen ostoskärryyn tai tuotteen siirtämisen kassalle. `Kauppa`-luokan testeissä ainoa merkityksellinen `Asiakas`-luokan toiminta on, että asiakas ottaa tuotteen hyllystä ja myöhemmin asettaa sen kassalle. Esimerkiksi, jos `Asiakas`-luokan toteutuksessa olisi bugi, jonka takia yhtä tuotetta ei aseteta kassalle ostoskärrystä. Jos `Asiakas`-luokka on toteutettu mock-oliona `Kauppa`-luokan testeissä tämä bugi ei vaikuta testeihin, koska `Asiakas`-luokan mock-olio ei käytä luokan omaa toteutusta kassalle tuotteiden asettamisesta, vaan asettaa kassalle testissä määritellyt tuotteet.

Mock-olioilla voidaan siis testata eristettynä luokkia, jotka riippuvat toisistaan ilman, että muutokset riippuvuuksissa vaikuttavat testattavaan luokkaan. Lisäksi mock-olioita voidaan käyttää tapauksissa, joissa riippuvuuksia ei ole vielä toteutettu. Jos edellisessä esimerkissä yksi tiimi toteuttaa Kauppa-luokan toiminnallisuuden ja toinen tiimi toteuttaa Asiakas-luokan toiminnallisuuden, voi ensimmäinen tiimi testata Kauppa-luokkaa ennen kuin toinen tiimi on saanut Asiakas-luokan toteutettua käyttämällä Asiakas-luokan ilmentymän sijaan mock-oliota.

Mock-olioita käytettäessä testauksen rakenne on yleensä seuraavanlainen:

- luo mock-oliot,
- alusta mock-olioiden tila,
- aseta mock-olioiden odotetut toiminnot tai tila testien jälkeen,
- suorita testattava koodi käyttäen mock-olioita,
- tarkasta mock-olioiden tila ja niille suoritetut toiminnot.

Ensiksi mock-oliot luodaan ja niiden tila alustetaan halutuksi. Seuraavaksi määritellään, mikä on mock-olioiden odotettu tila testien jälkeen ja mahdollisesti mitä operaatioita niille on suoritettu testien aikana. Tämän jälkeen suoritetaan itse testit käyttäen mock-olioita testattavan komponentin parametreina. Lopuksi tarkastetaan, että mock-olioiden tila vastaa odotuksia.

Mockaukseen liittyy myös metoditynkien (**stub**) luominen. Tynkä on versio metodista, joka metodin suorituksen sijaan palauttaa aina jonkin tietyn arvon. Tyngän palautusarvo voi riippua syötteestä ja metodi voidaan suorittaa joillakin syötteillä. Esimerkiksi listalle voidaan määrittää tynkä, joka palauttaa indeksillä yksi arvon ”yksi” ja muilla indekseillä listassa olevan arvon. Metodi-tyngillä voidaan muun muassa nopeuttaa suoritusta, jos jokin arvo saadaan esimerkiksi syötteenä tai metodin suoritus on hidasta, silloin kun metodin suorituksella ei ole merkitystä testin kannalta tai jos testissä halutaan käyttää jotakin tiettyä paluuarvoa riippumatta ohjelman tilasta. Tyngillä voidaan myös testata virhetilanteita, joiden toistaminen on vaikeaa.

2.5 Testauksen automatisointi

Testauksen automatisoinnissa, eli testiautomaatiossa, tavoitteena on, että suoritettavat testit suoritetaan tietokoneen toimesta [27]. Testiautomaation tarpeellisuus nousee, kun testattavien komponenttien ja rajapintojen määrä kasvaa. Pelkästään yksittäisen luokan tai metodin testauksessa testitapauksien määrä voi olla hyvin suuri erilaisten suorituspolkujen ja rajatapausten johdosta.

Esimerkiksi, jos testattavana on metodi, joka ottaa kokonaisluvun parametrina, tulee pelkästään parametristä vähintään kolme testitapausta:

- luku on positiivinen,
- luku on nolla,
- luku on negatiivinen.

Näiden kolmen tapauksen lisäksi voi olla tarpeen testata tapaukset, joissa luku on muuttujan, esimerkiksi 32-bittisen etumerkillisen kokonaisluvun (Int32), maksimi- ja minimiarvot. Näiden testitapausten päälle voi tulla metodin

toiminnasta riippuvia tapauksia, muuttuuko metodin toiminta merkittävästi, jos parametri on vaikkapa kymmenen tai tuhat, entä miljoona?

Metodin suorituspolut ovat kaikki ne eri tavat, joilla metodi voidaan käydä läpi. Yksinkertaisin esimerkki haarautuvista suorituspoluista on If-Else rakenne, jossa suoritetaan jompikumpi haara jonkin ehdon perusteella. Metodia testattaessa tulisi jokainen mahdollinen suorituspolku käydä läpi eri testeissä. Joten, jos esimerkiksi parametrista saadaan viisi tapausta ja metodilla on viisi erilaista suorituspolkua. Tällöoin testitapauksia on yhteensä kaksikymmentäviisi. Jos testaajan pitäisi suorittaa ohjelma jokaisella näillä tapauksilla testaukseen kuluu huomattavasti aikaa.

Ratkaisuna on automaatiotestaus. Tietokone voi suorittaa yksinkertaisia testitapauksia huomattavasti nopeammin kuin testaaja ja tietokone voi suorittaa testit huomattavasti eristetymmin, jolloin mahdolliset virheet on helppo kohdentaa. Testiautomaatiossa on käytössä jokin testien suoritin (**test driver**), joka ohjelmallisesti suorittaa yksittäiset testit ja raportoi testaajalle testien tulokset.

Testiautomaatio voidaan lisäksi yhdistää ohjelmistokehitysprosessiin erilaisilla työkaluilla. Esimerkiksi Maven-käännöstyökalulle voidaan määrittää tavoite `test`, jolloin ohjelman käännöksen yhteydessä työkalu suorittaa automaattisesti ohjelmalle määritellyt testit.

Testiautomaation toteutus vaihtelee käsiteltävän testaustason perusteella. Komponentti- ja integraatiotestit on usein yksinkertaista suorittaa käännöksen yhteydessä ja ne on helppo toteuttaa yksinkertaisina ohjelmallisilla testeillä. Regressiotestien suoritus automaattisesti on usein samankaltaista kuin komponentti- tai integraatiotestaus. Järjestelmä- ja hyväksyntätestien automatisointi on hieman monimutkaisempaa ja vaati usein erikoistuneempia työkaluja.

2.6 Testien valinta

Testimäärän rajoittamisessa tavoitteena on minimoida ohjelmoijan testaamiseen käyttämä aika ja samaan aikaan pitää huoli, että testejä suoritetaan mahdollisimman paljon ja mahdollisimman usein. On oleellista, että kaikki ohjelmakoodiin ja muihin resursseihin tehdyt muutokset testataan, ennen kuin ne voivat siirtyä tuotantoon. Rajoittamisen motivaationa on

Automaatiotestauksella saavutettu testausprosessin nopeus ei ole aina riittävää. Kaikkien muutosten kohdalla koko ohjelmiston testaaminen ei ole välttämättä mielekästä tai edes tarpeellista. Jos kehittäjä muuntaa yhtä komponenttia on muiden komponenttien yksikkötestien suorittaminen tarpeetonta. Testausprosessin pituutta ja kehittäjään kohdistuvaa kuormitusta voidaan vähentää määrittelemällä erilaisia testisarjoja (**test-suite**), joissa suoritetaan vain rajoitettu määrä testejä.

Toinen tapa nopeuttaa testausta on siirtää osa testauksesta pois kehittäjän koneelta. Perinteisesti kehittäjä suorittaa testisarjan ohjelmiston

käännöksen yhteydessä ja voi jatkaa työskentelyä vasta, kun koko testisarja on suoritettu. Jos osa testeistä siirretään esimerkiksi jatkuvan integraation (**Continuous Integration**) palvelimelle, ei kehittäjän tarvitse keskeyttää työskentelyä testien suorituksen ajaksi ja mahdollisesti hyvin pitkäkestoinen täysi testisarja suoritetaan palvelimella.

3 Tutkimusasetelma ja menetelmät

Tässä tutkielmassa tarkastellaan erilaisia ohjelmointialalla yleisesti käytössä olevia testauskehyksiä ja arvioidaan niitä kohdeyrityksen käyttötarkoituksen ja tarpeiden valossa. Yrityksen käyttötarkoitusta varten luodaan arviointikriteeristö, jonka pohjalta testauskehyksistä pyritään löytämään yrityksen tarpeita parhaiten vastaava ja sen ohjelmistokehitysprosessiin parhaiten sopiva testauskäytäntö.

Arviointia varten määritellään käyttötapaus ohjelmistoprosessista ja luodaan arviointikriteeristö, jonka perusteella eri testauskehyksien soveltuvuutta arvioidaan. Erityishuomiota arvioinnissa kiinnitetään siihen, miten helposti vaihtoehdot ovat integroitavissa olemassa olevaan Maven-ympäristöön. Lisäksi huomiota kiinnitetään eri testaustasojen saatavuuteen.

Lopuksi määritellään tutkimuskysymykset, joihin testauskehyksien arvioinnin pohjalta vastataan.

3.1 Kohdeyrityksen kuvaus

Kohdeyrityksen ohjelmistokehityksessä on pieni tiimi, joka kehittää ohjelmistoja asiakkaille ja yrityksen sisäiseen käyttöön. Sisäisessä käytössä olevia ohjelmia käyttävät yrityksen asiantuntijat, jotka eivät ole ohjelmistoalan ammattilaisia.

Ohjelmistoja kehitetään iteratiivisessa prosessissa. Jonkin asiakastarpeen tai tuoteidean pohjalta kehitetään alustava suunnitelma ohjelmistolle. Itse tuotekehitys tapahtuu pääasiallisesti asiakasprojekteissa, joissa tavoitteena on kehittää valmis ohjelmisto. Projektin aikana voidaan ohjelmistoista julkaista väliversioita, joita asiakas voi testata ja antaa palautetta toteutetuista ominaisuuksista ja selkeästi puuttuvista ominaisuuksista. Erillistä testausvaihetta ei ole, vaan testaus sisältyy ohjelmiston toteutusvaiheeseen. Testaus on tähän mennessä toteutettu lähinnä manuaalisesti ja tavoitteena olisi siirtyä automaattiseen testaukseen.

Ohjelmistokehitysprosessia seuraa ylläpito, jossa kehitystiimi vastaa ohjelman toimintaan liittyvästä ylläpidosta. Jos ohjelmisto sijaitsee yrityksen hallinnoimalla palvelimella, kehitystiimi vastaa myös palvelimen ylläpidosta. Ylläpitovaiheessa esiin tulevat bugit korjataan osana ylläpitoa, jos ilmenee tarve merkittäville muutoksille ohjelmistossa tai uusille ominaisuuksille aloitetaan niitä varten uusi ohjelmiston jatkokehitysprosessi.

Ohjelmiston elinkaaren aikana siitä julkaistaan useita versioita, joten on oleellista varmistaa ohjelmiston määritysten mukainen toiminta jokaisen version yhteydessä. Tällä hetkellä prosessiin ei kuulu automaatiotestausta, vaan testaus suoritetaan manuaalisesti. Manuaalinen testaus on hyvin aikaa vievää ja eritysesti vanhojen toiminnallisuuksien testaus jää usein vähemmälle jatkokehitysvaiheessa. Prosessiin tulee liittää automaatiotestaus, jonka toteutukseen käytettäviä ohjelmistokehyksiä tässä tutkielmassa tarkastellaan ja arvioidaan.

3.2 Tutkimuksen lähtökohdat

Tutkielmassa vertaillaan yleisesti ohjelmistoalalla käytössä olevia testauskehyksiä. Vertailuun valitaan testauskehyksiä, jotka soveltuvat yrityksen käyttötapaukseen. Ensiksi määritellään käyttötapaus, sitten käyttötapauksen perusteella muodostetaan arviointikriteerit, joiden mukaan testauskehykset arvioidaan.

Tarkasteltavat testauskehykset valitaan hakemalla Google-hakukoneella suosittuja Java testauskehyksiä ja rajaamalla tuloksista artikkeleita ja listoja, jotka ovat julkaistu aikaisintaan vuonna 2018. Näin löydetään ammattimaiseen käyttöön suositeltuja ja suosittuja testauskehyksiä, jotka ovat tuoreessa käytössä. Löydetyistä artikkeleista ja listoista valitaan käyttötapaukseen sopivat testauskehykset.

Seuraava lista kuvaa kohdeyrityksen ohjelmistokehityksen tyypilliset reunaehdot:

1. Java-ohjelmisto, jossa käytetään standardikirjastojen lisäksi kolmansien osapuolien avoimen lähdekoodin kirjastoja sekä kohdeyrityksen omia kirjastoja,

2. Ohjelmisto voi muodostua useasta ohjelmasta, jotka voivat olla sekä työpöytäsovelluksia että palvelinohjelmistoja.

3. Ohjelmistoilla on useita kehittäjiä ja ylläpitäjiä. Osa kehittäjistä ei ole enää yrityksen palveluksesta.

4. Ohjelmalla on graafinen käyttöliittymä joko verkkosivun tai työpöytäsovelluksen muodossa. Verkkosivut on toteutettu html-sivuina, joissa käytetään PrimeFaces kehystä. Työpöytäsovelluksien käyttöliittymä on toteutettu Swing-kehyksellä.

5. Ohjelma lukee syötettä tiedostoista sekä verkon yli eri rajapinnoista. Osa rajapinnoista voi olla kolmansien osapuolien hallinnoimia.

Yllä olevan käyttötapauksen pohjalta on luotu käyttötapauksen kriteerit, joiden pohjalta arvioidaan testauskehysten soveltuvuutta yrityksen käyttöön:

1. Yhteensopivuus Maven-ympäristön kanssa.

Kohdeyrityksen ohjelmistokehityksessä käytetään Maven-projekteja Java-ohjelmien kehityksessä. On oleellista, että käytettävä testauskehys on yhteensopiva käytetyn ympäristön kanssa.

2. Eri testaustasojen saatavuus.

Testauksen tulee olla mahdollisimman kattavaa. Käyttöönotettavat testauskehykset tulee valita siten, että valituista testauskehyksistä saadaan mahdollisimman moni testaustaso käyttöön.

3. Käytettävyys:

Testauskehyksen käytettävyys jakautuu kahteen osaan:

3.1. Kuinka helppokäyttöinen testauskehys on.

Testauskehyksen helppokäyttöisyys tarkoittaa, että testien kirjoittaminen on mahdollisimman nopeaa ja vaivatonta. Tällöin kehittäjä kirjoittaa mahdollisimman paljon testejä ja testauksesta saadaan mahdollisimman suuri hyöty irti.

3.2. Kuinka helppoa testauskehyksen käyttöönotto on.

Jos testauskehyksen käyttöönotto on monimutkaista kasvattaa se projektin aloituksessa tapahtuvaa konfigurointia. Jos vaadittu konfigurointi on mittavaa aiheutuu siitä ylimääräisiä kustannuksia ja viivästyksiä projektille.

4. Testidatan syöttäminen.

Testattaessa ohjelmiston ominaisuuksia tulee testit suorittaa mahdollisimman realistisella syötteellä. Osa ohjelmista voi käyttää hyvinkin monimutkaista syötettä ja voi olla tarpeellista antaa testeille esimerkiksi konfiguraatiotiedostoja syötteenä.

5. Raportoinnin helppous.

Testien tuloksista tulee saada selkeä tieto. Testien epäonnistuessa on oleellista tietää mikä testi epäonnistui ja miksi. Testiraportit ovat oleellisia sekä ohjelmiston kehittäjälle että ohjelmistoprojektin hallinnolle.

3.3 Tutkimuskysymykset

Arviointikriteeristöä hyödyntäen tutkielmassa vastataan seuraaviin tutkimuskysymyksiin:

RQ1. Minkälaisia testauskehyksiä on käytössä ohjelmistonkehitysalalla?

Automaatiotestaus suoritetaan erilaisia testauskehyksiä hyödyntäen. Selvitämme, minkäalaisia testauskehyksiä on saatavilla Java-ohjelmistojen testaukseen. Keskitymme tarkastelemaan yleisesti ohjelmistokehitysalalla käytössä olevia kehyksiä, jolloin voimme olla varmoja, että tarkasteltavat kehykset ovat luotettavia ja niiden toiminta on vakaata.

RQ2. Miten erilaiset testauskehykset soveltuvat kohdeyrityksen ohjelmistokehitysprosessiin?

Tarkasteltavia testauskehyksiä arvioidaan niiden soveltuvuudessa kohdeyrityksen ohjelmistokehitysprosessiin. Varmistamme, että testauskehykset soveltuvat yrityksen käyttämään prosessiin ja niiden tarjoamasta toiminnallisuudesta on arvoa prosessille ja ohjelmistoille.

RQ3. Mikä on kohdeyrityksen tarpeisiin parhaiten sopiva tapa integroida testaus ohjelmistokehitysprosessiin?

Arvioimme tarkasteltujen testauskehysten ja aiempien kysymysten pohjalta parasta tapaa integroida testaus kohdeyrityksen ohjelmistokehitysprosessiin. Erityistä huomiota kiinnitetään yrityksen prosessin erityistarpeisiin ja testauskehysten tarjoamaan toiminnallisuuteen.

3.4 Vertailukriteerit

Testauskehyksiä vertailtaessa kiinnitetään huomiota seuraaviin kriteereihin:

1. Tarjotut testaustyypit.

Mitä toiminnallisuutta testauskehys tarjoaa prosessiin, eli minkälaisia testejä sen avulla voidaan toteuttaa.

2. Sijoittuminen ohjelmistokehitysprosessiin.

Missä vaiheessa ohjelmistokehitysprosessia testauskehyksen käyttö tapahtuu. Tämän avulla tiedetään, milloin kehyksellä suoritettava testaus pitää olla suunniteltuna ja milloin kehyksellä suoritettavasta testauksesta saadaan tuloksia.

3. Testauksen automatisointi.

Miten testauskehystä voidaan automatisoida. Tähän sisältyy automaatiotestien suorituksen lisäksi automaation vaatima esityö. Esimerkiksi konfiguraatiotiedostojen luominen ja rakenne.

4. Suoritettavien testien valinta.

Suoritettavien testien ja testiluokkien jaottelu erilaisiin testisarjoihin. Eri testisarjoja voidaan hyödyntää, jos tilanteesta riippuen tarvitaan vain nopea osittainen testaus tai koko ohjelmiston täydellinen testaus.

5. Maven-integraatio.

Maven-integraatio on oleellinen kohdeyrityksen ohjelmistokehitysprosessissa. Yrityksen Java-ohjelmistoja kehitetään käyttäen Maven-käännöstyökalua, joten on oleellista, että käytettävät testauskehykset ovat yhteensopivia sen kanssa.

6. Lisenssi.

Ohjelmistokehityksessä käytettävillä työkaluilla ja kirjastoilla on useita erilaisia lisenssejä, jotka asettavat kehitettävälle ohjelmistolle ja ohjelmistokehitysprosessille rajoitteita ja vaatimuksia. Lisenssiehdot saattavat vaatia lisenssin ostamista tai tarjota käytettävän kehyksen ilman erillistä korvausta,

mutta voivat vaatia maininnan kyseisestä lisenssistä. On oleellista huomioida nämä vaatimukset ja varmistaa lakiteknisistä syistä lisenssin mukainen toiminta ennen ohjelmistokehityksen aloitusta.

Näiden kriteerien avulla tuodaan esille kohdeyrityksen ohjelmistokehitysprosessin kannalta merkityksellisiä ominaisuuksia testauskehyksissä.

3.5 Tiedonhakuprosessi

Testauskehyksiä haettiin Google-hakukoneesta hakutermillä ”java test frameworks”. Hakutulokset olivat pääosin erilaisia artikkeleita, joissa listattiin eri testauskehyksiä, joita suositeltiin Java-kehitykseen. Hakutuloksista valittiin tuoreita, vuosina 2018-2020 julkaistuja artikkeleita.

Hakutuloksista valittiin tarkasteltavaksi seuraavat artikkelit:

- Top 5 Java Test Frameworks for Automation in 2019 [18],
- Top 10 Testing Frameworks and Libraries for Java Developers [17],
- Best Java Unit Testing Frameworks [4],
- Top 10 Java Automation Test Frameworks and Libraries You Can Learn in 2020 [16],
- 25 Best Java Testing Frameworks And Tools For Automation Testing (Part 3) [2],
- 11 top open-source test automation frameworks: How to choose [1],
- What are the Latest Java Test Frameworks for 2019? [19].

Tarkastelluista artikkeleista kerättiin testauskehykset, jotka oli listattu useammassa artikkelissa. Tässä tutkielmassa ei tarkastella Behavior Driven Development(BDD) testausta, joten BDD-testauskehyksiä ei valittu tarkasteltavaksi. Tarkasteltavat testauskehykset on listattu seuraavassa kappaleessa.

4 Tulokset

Tiedonhakuprosessissa tarkastelluista artikkeleista valittiin seuraavat testauskehykset, testauskehykset on myös jaoteltu luokkiin niiden tarjoamien toiminnallisuuksien perusteella:

- Yksikkötestaus
  - JUnit [9]
  - TestNG [15]
  - Spock Framework [14].

- Mockaus
  - Mockito [10]
  - EasyMock [6].
- UI testaus
  - Selenium [12]
  - Selenide [11].

Näiden lisäksi artikkeleista löydettiin muihin kriteereihin sopivia Behavior Driven Development(BDD) testauskehyksiä: JBehave [8], Cucumber [5] ja Serenity [13]. kohdeyrityksen ohjelmistokehitys ei seuraa BDD-prosessia, joten näitä ei käsitellä tässä tutkielmassa.

Testauskehykset tarkastellaan toiminnallisuuden mukaisessa järjestyksessä. Ensin tarkastellaan yksikkötestauskehykset, seuraavaksi mockaus-kehykset ja lopuksi käyttöliittymätestauskehykset. Jokaisen aliluvun alussa esitellään lyhyesti toiminnallisuuden perusteet, jonka jälkeen käydään läpi yksittäiset testauskehykset. Kehysten jälkeen käydään yhteenvetona toiminnallisuuden testuskehysten yhtäläisyydet ja eroavaisuudet. Testauskehysten jälkeen esitetään yhteenveto kaikista tarkastelluista testauskehyksistä vertailukriteerien suhteen.

4.1 Yksikkötestaus

Yksikkötestauskehykset ovat testauskehyksiä, joita käytetään yksikkötestien toteuttamiseen. Yksikkötestien lisäksi yksikkötestauskehyksiä voidaan käyttää muun muassa integraatio- ja regressiotestien toteutukseen. Teknisesti näissä tapauksissa testien toteutus ei eroa yksikkötesteistä. Erona on integraatiotestien tapauksessa testiobjektit, jotka ovat yksittäisen moduulin sijaan rajapintoja, joiden yhteistoimintaa tarkastellaan testissä. Regressiotesti voi olla yksikkö- tai integraatiotesti, jossa varmistetaan, että ohjelmiston toiminta ei ole muuttunut edellisen testauksen jälkeen.

4.1.1 JUnit

JUnit on yksikkötestauskehys Javalle. JUnit pohjautuu xUnit-kehykseen, joka on suosittu perhe yksikkötestauskehyksiä. XUnit toteutuksia löytyy yli 80 kielelle [20]. JUnitin nykyinen versio on 5.6.0 (JUnit 5). JUnit 5 muodostuu kolmesta osasta: JUnit Platform, JUnit Jupiter ja JUnit Vintage.

JUnit-testit tulisi kirjoittaa ohjelmiston kehityksen yhteydessä, ideaalissa tilanteessa testit kirjoitettaisiin ennen toiminnallisuutta. Testaus ja testien luominen sijoittuvat toteutus- ja testausvaiheeseen ohjelmiston elinkaaressa.

JUnit Platform on alusta testauskehysten ajamiseen. JUnit Platform toimii rajapintana ohjelmointityökalujen, kuten IDE:jen ja käännöstyökalujen,

```
public class JunitTest{

    private String myTestVariable;

    @BeforeEach
    static void init(){
        myTestVariable = "foo";
    }

    @Test
    static void myTest(){
        String temp = "foo";
        assertEquals(myTestVariable, temp);
    }

    @ParameterizedTest
    @ValueSource(strings = {"foo", "bar"})
    static void parameteredTest(String name){
        assertEquals(myTestVariable, name);
    }
}
```

Esimerkki 1: JUnit testiluokka.

ja Javan virtuaalikoneen välillä. JUnit Platform määrittelee TestEngine-rajapinnan, jolla voidaan toteuttaa testauskehyksiä, joita ajetaan alustalla. Alustan mukana toimitetaan konsoliapplikaatio testien suorittamiseen, tämän lisäksi suosituimmat IDE:t ja käännöstyökalut sisältävät JUnit Platform tuen.

JUnit Jupiter tarjoaa TestEnginen, jolla JUnit 5 testit kirjoitetaan. JUnit Vintage tarjoaa taaksepäin yhteensopivuuden vanhempien JUnit versioiden kanssa. JUnit Vintage tarjoaa TestEnginen, jolla voidaan suorittaa JUnit 4 ja JUnit 3 testejä. Keskitymme jatkossa ainoastaan JUnit Jupiterilla toteutettaviin testeihin.

JUnit testit toteutetaan luomalla testiluokka. Testiluokan metodeille annetaan erilaisia annotaatioita, joiden avulla testit konfiguroidaan. Esimerkiksi `@Test`-annotaatio tarkoittaa, että metodi on testi. Annotaatioilla voidaan myös määrittää apumetodeja, esimerkiksi `@BeforeEach`-annotaatio tarkoittaa, että metodi suoritetaan ennen jokaista testiä. JUnit tarjoaa testausta varten erilaisia metodeja, tärkeimpänä näistä on erilaiset `assert`-metodit, kuten `assertEquals()` ja `assertTrue()`. `Assert`-metodeille annetaan metodista riippuvat parametrit, joiden totuusarvo määritetään. Esimerkiksi `assertEquals()`-metodille annetaan kaksi samantyyppistä arvoa tai oliota.

Jos niiden arvo on sama testi onnistuu ja jos arvot eivät ole samat testi epäonnistuu. Jos yhdessä testimetodissa on useita `assert`-metodikutsuja testi onnistuu ainostaan jos kaikki `assert`-kutsut onnistuvat.

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.22.2</version>
      <configuration>
        <excludes>
          <exclude>OldTest.java<exclude/>
        </excludes>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Esimerkki 2: JUnit exclude-sääntö pom.xml-tiedostossa.

JUnit testit voidaan suorittaa joko manuaalisesti komentoriviltä tai IDE:n kautta, tai käännöstyökalun avulla. Maven-käännöstyökalulla konfiguroidaan JUnit-liitännäinen (plugin) käännösprosessin `test`-vaiheeseen. Oletuksena liitännäinen etsii kaikki Java-luokat, joiden nimi alkaa tai päättyy **Test** tai päättyy **Tests** tai **TestCase**. Lisäksi liitännäiselle voidaan määritellä pom.xml-tiedostossa `exclude` ja `include` sääntöjä, joiden perusteella oletustoiminnallisuudesta poiketaan lisäämällä tai poistamalla suoritettavien testiluokkien listasta määritellyt luokat.

JUnit on avointa lähdekoodia ja se tarjotaan käyttöön Eclipse Public License versio 2.0 [7] alaisuudessa. Lisenssi mahdollistaa ohjelman ilmaisen käytön myös kaupallisiin tarkoituksiin.

4.1.2 TestNG

TestNG on yksikkötestauskehys Javalle. TestNG on JUnitin inspiroima ja niillä on paljon yhteistä, esimerkiksi annotaatioiden käyttö testimetodien tunnistamisessa.

Kuten JUnitissa TestNG testit sijoittuvat toteutus- ja testausvaiheeseen.

TestNG testit luodaan kuten JUnitissa. Luodaan testiluokka, johon luodaan testimetodeja `@Test`-annotaatiolla. Testimetodeille voidaan määrittää annotaatiossa erilaisia attribuutteja. Esimerkiksi `groups`-attribuutilla voidaan määritellä testiryhmiä ja `enabled`-attribuutilla voidaan määrittää, suoritetaanko yksittäistä testiä vai ei riippumatta muista asetuksista. Testeille voidaan lisäksi määrittää parametreja `@Parameter`-annotaatiolla. Jos halu-

```
public class MyTest {
    @DataProvider(name = "test_data")
    public String[] createTestData(){
        return new String[] {
            "foo", "bar", "baz"
        }
    }

    @Test(groups = {"relevantTests"})
    public void testMethod(){
        assert 2 > 1;
    }

    @Test(dataProvider = "test_data")
    public void testArray(String name){
        assertEquals(name, "foo");
    }
}
```

Esimerkki 3: TestNG testiluokka.

taan käyttää monimutkaisempia parametreja, esimerkiksi kompleksoja Java-olioita voidaan käyttää `@DataProvider`-annotaatiota. Ensiksi luodaan metodi, jolle annetaan `@DataProvider`-annotaatio. `@DataProvider`-annotaatiolle annetaan attribuutti `name`, jota TestNG käyttää DataProviderin tunnisteena. Tämä metodi palauttaa parametrina käytettävän olion. Seuraavaksi luodaan testimetodi, jonka `@Test`-annotaatiolle annetaan `dataProvider`-attribuuttina aiemmin luodun DataProviderin nimi.

TestNG käyttää ehtojen tarkastamiseen Javan `assert`-avainsanaa. `Assert`-avainsanalle annetaan ehto, jolla on boolean-arvo. Jos arvo on epätosi `assert` heittää virheen `AssertionErrorException`, jonka TestNG tunnistaa ja tulkitsee epäonnistuneeksi testiksi. Assert avainsanan lisäksi TestNG:ssä voidaan käyttää JUnit-tyylisiä assert-metodeja kuten `assertEquals`.

Suoritettavat testit määritellään *testng.xml*-tiedostossa. Tiedostossa voidaan määrittää useita testisarjoja xml-tagilla `<suite>` eri testisarjoille voidaan määrittää eri tulostusasetukset `verbose`-attribuutilla. Testisarjan sisälle määritellään testit xml-tagilla `<test>`, testissä suoritettavat testit voidaan määrittää joko testiluokan tai Java-pakkauksen perusteella. Testiluokkaa käytetään xml-tagilla `<classes>`, jonka alle listataan xml-tageissa `<class>` yksittäiset Java-luokat, luokkien nimet voidaan määritellä myös käyttäen pakkauksen nimeä erityisesti, jos ohjelmistossa on useita samannimisiä luokkia eri pakkauksissa. Testille voidaan antaa Java-pakkaus xml-tagilla `<packages>`, jonka sisälle listataan xml-tageilla `<package>` yksittäiset Java-pakkaukset.

```
<!DOCTYPE suite SYSTEM
    "https://testng.org/testng-1.0.dtd">
<suite name="test_1" verbose="1" >
    <test name="class_test" verbose="1" >
        <classes>
            <class name="myTest" />
        </classes>
    </test>
    <test name="package_test" verbose="1" >
        <packages>
            <package name="fi.org.tests" />
        </packages>
    </test>
    <test name="tuned_test" verbose="1" >
        <groups>
            <run>
                <include name="relevantTests" />
                <exclude name="brokenTests" />
            </run>
        </groups>
        <classes>
            <class name="myAdditionalTest" >
                <methods>
                    <include name="testMethod" />
                </methods>
            </class>
        </classes>
    </test>
</suite>
```

Esimerkki 4: testng.xml tiedoston rakenne.

Yksittäisiä testimetodeja voidaan jättää suorittamatta käyttämällä xml-tageja `<include>` ja `<exclude>`, niiden sisään listataan joko xml-tagilla `<groups>` testiryhmät tai xml-tagilla `<methods>` yksittäisten metodien nimet.

TestNG testit voidaan suorittaa joko manuaalisesti komentoriviltä tai IDE-liitännäisellä, tai käännöstyökalun avulla. Maven-käännöstyökalulla suoritettaessa annetaan testng.xml tiedosto ja testiraportin tulostusnimi ja -hakemisto, Mavenille määritellään tavoite `testng` ja `testng:junit-report`, joista ensimmäinen suorittaa TestNG testit ja toinen tekee suoritetuista testeistä raportin.

TestNG on avointa lähdekoodia ja se on lisensoitu Apache 2.0 lisenssillä

[3]. Lisenssi mahdollistaa ohjelman käytön kaupallisiin tarkoituksiin.

4.1.3 Spock Framework

Spock Framework on määritelmäpohjainen testauskehys. Spock-testit kirjoitetaan Groovy-kielellä ja niissä tehdään metodeja määritelminä, esimerkiksi `def "pushing an element on the stack"()` olisi metodi, jossa pinoon asetetaan elementti.

Spock-testien määritelmäpohjaisuus mahdollistaa testien luomisen ohjelmiston määrittely- ja suunnitteluvaiheessa. Testitapaukset voidaan luoda heti kun komponenttien määritelmät ovat olemassa ja testimetodien toteutukset voidaan luoda ohjelmoinnin yhteydessä.

Spock-testit muodostuvat lohkoista. Eri lohkot hoitavat eri tehtäviä testissä ja niillä voi olla vaatimuksia sijainnista testissä tai edeltävistä ja seuraavista lohkoista. `Given`-lohkossa alustetaan testin tarvitsemat tiedot, `given`-lohko voi olla implisiittinen, joten sitä ei välttämättä tarvitse kirjoittaa lohkon alkuun. `When`- ja `then`-lohkot ovat aina pareissa, `when`-lohkossa määritellään jokin tapahtuma ja sitä vastaavassa `then`-lohkossa määritellään ehdot, jotka tulee toteutua. Jos jokin `then`-lohkossa oleva ehto ei toteudu, testi epäonnistuu. `Expect`-lohko yhdistää `when`- ja `then`-lohkojen toiminnan, `expect`-lohkossa voidaan tarkastaa jokin ehto ilman `when`-lohkossa määriteltyä tapahtumaa. `Cleanup`-lohkon tulee olla testin lopussa. Sen jälkeen voidaan määrittää ainoastaan `where`-lohkoja. `Cleanup`-lohkossa vapautetaan testissä käytetyt resurssit ja se suoritetaan aina testin lopuksi, vaikka testin aikana olisi tapahtunut virhe. `Where`-lohko voidaan määrittää ainoastaan testin viimeiseksi lohkoksi. `Where`-lohkossa voidaan määrittää data, jolla testi suoritetaan ja mahdollisesti data usealle suorituskerralle.

Spock-testit suoritetaan omalla JUnit TestEnginellä, testit voidaan suorittaa Mavenilla tai Gradlella. Maven-käännöstyökalua käytettäessä täytyy Spock Framework-riippuvuuden lisäksi määritellä Groovy-riippuvuus Groovy-koodin kääntämistä varten. Testien suoritus tapahtuu Maven-liitännäisellä `test`-vaiheessa. Suoritettavat testit valitaan automaattisesti luokkien nimen perusteella, kuten JUnitissa, tai ne voidaan määritellä Maven-Surefire-liitännäisen include- ja exclude-säännöillä.

Spock Framework on avointa lähdekoodia ja se on julkaistu Apache 2.0-lisensillä.

4.1.4 Yhteenveto

JUnit, TestNG ja Spock Framework tarjoavat kaikki yleisen testisarjojen suorittamisen. Kaikissa kolmessa on toteutettu erilaisten testiehtojen tarkastus.

TestNG on JUnitin inspiroima, joten niiden käyttämä testirakenne on hyvin samanlainen. Molemmat käyttävät annotaatioita testi- ja apumetodien

```
def "example test"() {
    given:
    def file = new File("/path")
    def object = new MyObject()
    file.createNewFile()

    expect:
    file.exists()

    when:
    object.someMethod("foo")

    then:
    object.someField == "foo"

    cleanup:
    file.delete()
}

def "test with where block"() {
    expect:
    Math.max(a, b) == c

    where:
    a << [1, 2, 3]
    b << [2, 2, 2]
    c << [2, 2, 3]
}
```

Esimerkki 5: Spock Framework testi.

tunnistamiseen ja niiden nimeämiskäytännöt ovat hyvin samanlaiset. Suurimmat erot niiden välillä tulevat testisarjojen määrittelyssä ja suorituksessa.

Spock Framework eroaa huomattavasti JUnitista ja TestNG:stä sen määritelmäpohjaisen rakenteen vuoksi. Toinen suuri ero Spock Frameworkin ja JUnitin sekä TestNG:n välillä on Spockin käyttämä Groovy-kieli ja testien lohkorakenne.

4.2 Mockaus

Mockauksessa korvataan olio, jonka toteutuksella ei ole testin kannalta merkitystä, mock-oliolla. Mock-olio ei sisällä olion toteutusta, vaan testin kannalta merkityksellisiä toimintoja. Mock-olio voi raportoida, mitä metodikutsuja

sille on suoritettu ja se voi palauttaa metodeillaan ennalta määrättyjä arvoja.

Mockauskehyksiä voidaan hyödyntää testauksessa tilanteissa, joissa testiobjekti kutsuu jotakin toista oliota, mutta toisen olion suorituksella ei ole testin kannalta merkitystä. Mock-olioilla voidaan tarkastaa, että testiolio ei tee odottamattomia kutsuja muihin olioihin. Mock-olioita voidaan hyödyntää silloin, kun testiobjekti on toteutettu, mutta kaikkia sen käyttämiä rajapintoja tai muita olioita ei ole vielä toteutettu tai niiden kutsuminen olisi liian aikaa vievää. Mock-olioilla voidaan myös toteuttaa vaikeasti testattavia tilanteita, kuten virheitä tai joitakin tarkkoja reaaliaikajärjestelmien tiloja. Esimerkiksi kellonaikoja tai päivämääriä voidaan testata mock-oliolla helposti verrattuna reaaliaikakelloon.

4.2.1 Mockito

Mockito on testauskehys, joka tarjoaa toiminnallisuuden mock-olioiden luomiseen.

Mockaus liittyy oleellisesti jonkin toisen testin suoritukseen, joten Mockito-testit eivät itsessään sijoitu mihinkään erityiseen ohjelmistokehitysprosessin vaiheeseen. Mockauksen edut ovat kuitenkin suurimmillaan projektin alussa, jolloin konkreettisia toteutuksia ei välttämättä ole vielä saatavilla. Lisäksi mock-olioita voidaan hyödyntää tilanteissa, joissa jokin ohjelman osa on muuttumassa huomattavasti ja siten soveltuvat erityisen hyvin integraatio- ja regressiotestaukseen.

Mockitolla voidaan luoda versio oliosta, jonka toiminnallisuutta voidaan muokata testin yhteydessä. Esimerkiksi voidaan luoda pino, joka palauttaa aina tietyn arvon riippumatta mitä kekoon on asetettu. Näin voidaan simuloida tilanteita ja ehtoja, joiden asettaminen tai tapahtuminen olisi käytännön tilanteessa epätodennäköistä tai työlästä.

Mockitolla voidaan myös verifioida, että mock-oliolle on tehty tietyt operaatiot. Näin voidaan varmistaa, että testattava toiminnallisuus ei tee ylimääräisiä tai ei-toivottuja operaatioita.

Mock-olio luodaan kutsumalla konstruktorimetodia `mock(Class class)`. Metodille annetaan parametriksi joko rajapinta tai konkreettinen luokka, esimerkiksi `mock(List.class)`. Tämän jälkeen mock-oliota voidaan käyttää kuten mitä tahansa oliota. Mock-oliolle voidaan määrittää palautusarvoja `when()`-metodilla. Esimerkiksi kutsumalla metodia `when(list.get(0)).thenReturn( "ensimmäinen")` lista `list` palauttaa arvon `"ensimmäinen"`, kun sen ensimmäistä alkiota haetaan riippumatta siitä, mikä ensimmäinen alkio todellisuudessa on.

Mock-oliolle tehdyt operaatiot voidaan tarkastaa `verify()`-metodilla. Verfiy varmistaa, että kyseistä metodia on kutsuttu tasan kerran annetulla parametrilla. Esimerkiksi `verify(list).add("kerran")` tarkastaa, että oliolle `list` kutsutaan `add`-metodia tasan kerran arvolla `"kerran"`. Jos verifyn ehto ei täyty testi epäonnistuu. Verify:lle voidaan antaa mock-olion lisäksi

parametri, joka määrittää kuinka monta kertaa kyseistä metodia kutsutaan kyseisellä parilla. Parametri voi olla esimerkiksi `never()`, jolloin varmistetaan, että kyseistä metodia ei kutsuta kertaakaan kyseisellä parametrilla, `times(int n)` varmistaa, että kyseistä metodia kutsutaan `n`-kertaa kyseisellä parametrilla ja `atLeast(int n)` sekä `atMost(int n)` varmistavat, että kyseistä metodia kutsutaan enintään tai vähintään `n`-kertaa kyseisellä parametrilla. Lopuksi voidaan tarkastaa, että oliolle ei ole suoritettu enempää operaatioita `verifyNoMoreInteractions()`-metodilla.

```
public void MockitoTest() {
    List list = mock(List.class);

    when(list.get(10)).thenReturn("kymmenen");

    list.add(1);
    list.add(2);

    verify(list).add(1);
    verify(list).add(2);
    verify(list).add(10); //This test fails.
}
```

Esimerkki 6: Testi Mockiton mock-oliolla.

Mockito ei itsessään tarjoa testien suoritusta, vaan se on tarkoitettu käytettäväksi yhdessä muiden testauskehysten kanssa. Mockito voidaan asentaa Mavenin avulla, mutta Mockitolle ei ole tarjolla omaa Maven-liitännäistä.

Mockito on julkaistu MIT-lisenssillä ja on siten avoimen lähdekoodin ohjelma.

4.2.2 EasyMock

EasyMock on testauskehys, joka on tarkoitettu mock-olioiden luomiseen. Kuten Mockitossa EasyMock-testit ovat oleellisesti sitoutuneita muihin testeihin. Mockauksen edut korostuvat projektin alkuvaiheessa ja integraatiotestauksessa.

EasyMockilla voidaan luoda mock-olioita rajapinnoista tai konkreettisista luokista. Luodulle mock-oliolle talletetaan oletettu toiminnallisuus, jonka jälkeen suoritetaan testikoodi ja lopuksi mock-olio verifioidaan. Mock-olioita käytettäessä voidaan hyödyntää JUnit tyylisiä annotaatioita, esimerkissä olevat `@Test`- ja `@Rule`-annotaatiot ovat JUnit annotaatioita. Mock-olio luodaan `@Mock`-annotaatiolla, annotaation sijaan mock-olio voidaan luoda Mockiton tyylisellä `mock(Class class)` kutsulla, esimerkiksi `mock = mock(MockClass.class)`. Testattava luokka luodaan `@TestSubject`-annotaa-

```
public class EasyMockTest {
    @Rule
    public EasyMockRule rule = new EasyMockRule(this);

    //TestClass uses MockClass internally
    @TestSubject
    private TestClass testClass = new TestClass();

    @Mock
    private MockClass mock;

    @Test
    public void exampleTest() {
        mock.actionPerformed("test");
        replay(mock);
        testClass.method("test", 1);
        verify(mock);
    }
}
```

Esimerkki 7: Testi EasyMockin mock-oliolla.

tiolla, testattava luokka voidaan luoda ilman annotaatiota, jolloin mock-olio tulee asettaa testattavalle oliolle manuaalisesti. Esimerkissä oletetaan, että `TestClass`-luokka käyttää `MockClass`-luokan oliota sisäisesti ja `method()`-kutsu kutsuu `MockClass`-luokan `actionPerformed()`-metodia. Esimerkissä oleva `@Rule`-annotoitu `EasyMockRule` kertoo JUnit:lle kuinka mock-oliota tulee käyttää tässä luokassa.

Mock-oliolle talletetaan toiminnallisuus kutsumalla mock-olion niitä metodeja, joita testissä pitäisi kutsua, niillä parametreilla, joita testeissä käytetään. Kun oletettu toiminnallisuus on talletettu, kutsutaan `replay(mock)`-metodia joka asettaa mock-olion testaus-tilaan. Testaus-tilassa ajetaan testikoodi testattavalle oliolle. Lopuksi verifioidaan, että mock-oliolle suoritettiin vain ne kutsut, jotka sille oli ohjelmoitu, eikä mitään ylimääräisiä `verify(mock)`-metodilla.

EasyMock ei itsessään tarjoa kehystä testien suorittamiseen, vaan se on tarkoitettu käytettäväksi jonkin testejä suorittavan testikehyksen kanssa, esimerkiksi JUnit. EasyMock voidaan asentaa Mavenin avulla, mutta sille ei ole omaa Maven-liitännäistä.

EasyMock on julkaistu Apache 2.0 lisenssillä ja on siten avointa lähdekoodia ja käytettävissä ilmaiseksi myös kaupallisiin tarkoituksiin.

4.2.3 Spock Framework

Spock Framework sisältää oman toteutuksen mock-olioiden luomiseen.

```
class PublisherSubscriberTest extends Specification {
    Publisher publisher = new Publisher()
    Subscriber subscriber = Mock()

    def setup() {
        publisher.subscribers.add(subscriber)
    }

    def "test using mock-object"() {
        when:
        publisher.send("hello")

        then:
        1 * subscriber.receive("hello")
    }
}
```

Esimerkki 8: Testi Spock Frameworkin mock-oliolla.

Mock-oliot luodaan kutsumalla `Mock(class)`-metodia. `Mock()`-metodille ei tarvitse antaa luokkaa, jos mock-olio sijoitetaan tyypitettyyn muuttujaan `def` muuttujan sijaan. Tämän jälkeen mock-oliolle suoritettuja kutsuja voidaan verifioida `then`-lohkossa, kutsumalla testattavaa metodia `when`-lohkossa.

Oletamme esimerkissä 8, että `Publisher`-luokalla on metodi `send(String str)`, joka lähettää annetun viestin kaikille kyseiseen `Publisher`:iin liitetyille `Subscriber`-olioille kutsumalla niiden `receive(String str)`-metodia.

Spock Framework sisältää oman TestEnginensä, joten se ei tarvitse erillistä testikehystä testien suorittamista varten.

4.2.4 Yhteenveto

Mockito, EasyMock ja Spock Framework tarjoavat kaikki perustoiminnallisuuden mockaukseen. Mock-olioiden luomisen ja niille suoritettujen toimintojen tarkastamisen. Spock Framework eroaa Mockitosta ja EasyMockista siinä, että Spock Framework sisältää itsessään testien suorittamiseen tarvittavan toiminnallisuuden, kun Mockito ja EasyMock pitää yhdistää johonkin toiseen testien suorittimeen testausta varten.

Kaikkien kolmen syntaksi ja testitapausten rakenne eroavat toisistaan. EasyMock mahdollistaa JUnit tyylisten annotaatioiden käytön. EasyMockissa toiminnallisuus verifioidaan ensin tallettamalla odotetut operaatiot, sen jälkeen suorittamalla testissä tapahtuvat operaatiot ja lopuksi mock-olio

verifioidaan yhdellä funktiokutsulla. Mockitossa suoritetaan ensin testissä tapahtuvat operaatiot ja vasta lopuksi syötetään verifioitavat operaatiot mock-oliolle. Spock Frameworkin mockaus hyödyntää sen lohkorakennetta ja toimii muuten hyvin samalla tavalla kuin Mockitossa ja EasyMockissa.

4.3 UI-testaus

UI-testauksessa pyritään varmistamaan, että ohjelmiston käyttöliittymä toimii määritysten mukaisesti. UI-testauksessa simuloidaan käyttäjän toimintaa, joten funktiokutsujen sijaan toiminnot suoritetaan käyttöliittymän elementeillä.

4.3.1 Selenium

Selenium on käyttöliittymätestauskehys, joka toimii web-selainten kanssa. Selenium on siis web-applikaatioiden testaukseen tarkoitettu testauskehys. Seleniumia voidaan käyttää web-applikaation järjestelmä- ja regressiotestaukseen. Selenium on saatavilla useille eri kielille, muun muassa Java, Python, C#, Ruby, JavaScript ja Kotlin.

Selenium-testit sijoittuvat ohjelmistokehitysprosessissa testaus- ja hyväksyntävaiheeseen. Yksinkertaisemman toiminnallisuuden testaus on paljon kannattavampaa yksikkötesteillä kuin UI-testeillä ja UI-testauksessa halutaan varmistaa ohjelmiston toimintaa loppukäyttäjän näkökulmasta.

Seleniumin ytimessä on Selenium WebDriver. WebDriver simuloi selaimen käyttöä samalla tavalla kuin käyttäjä käyttäisi sitä. Tavoitteena on, että testaajan ei tarvitse tietää, minkälaisella elementillä jokin toiminto tehdään, ainoastaan jokin tunniste siihen, esimerkiksi elementin id tai nimi. Tällöin testejä ei tarvitse muuttaa jos jokin toiminto siirretään toisenlaiseen elementtiin, esimerkiksi napista pudotusvalikkoon.

Seleniumin WebDriver tarjoaa erilaisia metodeja selaimen manipulointiin. WebDriver on rajapinta, jolle löytyy toteutuksia useille selaimille. Osa selaintoteutuksista on suoraan Seleniumin tarjoamia, kuten Firefox ja Internet Explorer, osa selaintoteutuksista on kolmansien osapuolien tarjoamia, kuten Google Chrome ja Microsoft Edge. WebDriverilla voidaan esimerkiksi avata jokin verkkosivu ja tämän jälkeen sillä voidaan suorittaa käyttäjälle tarjolla olevia toimintoja, kuten lukea sivun nimi tai navigoida sivuhistoriaa. Lisäksi sivulta voidaan etsiä elementtejä tietyillä tunnisteilla ja antaa syötettä kenttiin.

Selenium ei itsessään tarjoa testien automaatiota, mutta Selenium testit ovat vain metodeja luokissa ja niiden suoritus voidaan automatisoida muilla tavoin. Selenium testit voidaan esimerkiksi toteuttaa Java-luokkina, jotka suoritetaan Java-ohjelmana tai mahdollisesti JUnit testien yhteydessä. Toinen mahdollisuus on toteuttaa testit Python-skriptinä, jonka suoritus voidaan automatisoida ohjelmiston käännöksen yhteyteen.

```
public class SeleniumTest {
  public void exampleTest() {
    System.setProperty("webdriver.chrome.driver",
        "G:\\chromedriver.exe");
    WebDriver driver = new ChromeDriver();
    String exampleUrl = "www.mydomain.com";
    String username = "user1";
    String password = "password1";

    driver.get(baseUrl);

    WebElement elem = driver.
        findElement(By.id("usrName"));
    elem.sendKeys(username);
    elem = driver.findElement(By.id("passwd"));
    elem.sendKeys(password);

    elem = driver.findElement(By.id("login"));
    elem.click();

    Assert,assertTrue("User␣should␣be␣logged␣in.",
        driver.getTitle.equals(username));
    driver.quit();
  }
}
```

Esimerkki 9: Selenium testi.

Selenium WebDriver on saatavilla Mavenin kautta. Tällöin Maven huolehtii ohjelmakirjastojen lataamisesta ja versioinnista. Seleniumille ei ole tarjolla Maven-liitännäistä, joka mahdollistaisi testien suorituksen käännöksen yhteydessä.

Selenium on avoimen lähdekoodin ohjelma, joka on lisensoitu Apache 2.0 lisenssillä.

4.3.2 Selenide

Selenide on käyttöliittymätestauskehys, joka pohjautuu Selenium WebDriveriin. Samoin kuin Selenium, Selenidea käytetään web-sovellusten käyttöliittymätestaukseen. Toisin kuin Selenium, Selenide on tarjolla ainoastaan Javalla.

Selenide-testit sijoittuvat testaus- ja hyväksyntävaiheeseen samoista syistä kuin Selenium-testit.

Selenide pyrkii yksinkertaistamaan Seleniumin tarjoamaa toiminnallisuutta. Selenide kapseloi WebDriverin toiminnan ja testejä kirjoitettaessa ei tarvitse kiinnittää huomiota esimerkiksi siihen, että selain käynnistetään ja suljetaan. Seleniden testit ovat ytimekkäämpiä kuin Seleniumin, elementtien haku tapahtuu `$`-metodilla tai joukko elementtejä `$$`-metodilla. Elementtejä voidaan hakea joko CSS-tunnisteella tai Seleniumin `By`-oliolla. Kun elementti on haettu sen tilaa voidaan tarkastella metodeilla, esimerkiksi `text()`-metodi palauttaa elementin tekstin. Elementtien toiminto-metodeilla voidaan suorittaa käyttäjän tekemiä toimintoja, esimerkiksi `click()`-metodi simuloi elementin klikkaamista ja `setValue()`-metodilla voidaan antaa tekstikentälle syötettä.

```
public void SelenideTest() {
    String exampleUrl = "www.mydomain.com";
    String username = "user1";
    String password = "password1";
    open(exampleUrl);

    $("#usrName").setValue(username);
    $("#passwd").setValue(password);

    $("#login").click();

    $("#user").shouldNotBe(empty);
}
```

Esimerkki 10: Selenide testi.

Selenide testitapaukset toteutetaan `should()`-metodeilla. Näitä metodeja

ovat `should()`, `shouldBe()`, `shouldHave()`, `shouldNot()`, `shouldNotBe()` ja `shouldNotHave()`. Metodeille annetaan jokin ehto ja metodin valinta riippuu käytettävästä ehdosta. Esimerkiksi jos halutaan tarkistaa onko id:llä `"user"` oleva kenttä tyhjä käytetään `$("#user").shouldBe(empty)`-metodia ja ehtoa. Jos taas halutaan tarkastaa, onko samalla kentällä CSS-luokka `"input"`, käytetään `$("#user").shouldHave(cssClass("input"))`-metodia ja ehtoa.

Kuten Selenium, Selenide ei itsessään tarjoa toiminnallisuutta testien automaattiseen suoritukseen. Selenide testit ovat Java-metodeja, joten ne voidaan suorittaa joko Java-ohjelmana tai jonkin automaatiotestikehyksen yhteydessä. Selenide riippuvuus voidaan asentaa Mavenin kautta.

Selenide on avoimen lähdekoodin ohjelma, joka on julkaistu MIT-lisenssillä. MIT-lisenssi mahdollistaa ohjelman käytön ja jakelun ilmaiseksi myös kaupallisissa tarkoituksissa.

4.3.3 Yhteenveto

Selenium ja Selenide ovat UI-testauskehyksiä, jotka on tarkoitettu web-selainten testaamiseen. Selenide pohjautuu Seleniumiin, joten toiminnallisesti niillä ei ole suuria eroja. Seleniumissa on tuki useammalle kielelle kun taas Selenide on tarjolla ainoastaan Javalla, tämä tarkoittaa sitä, että Selenide-testit pitää ajaa Java-ympäristössä vaikka niillä voidaan testata muillakin kuin Javalla tehtyjä web-sovelluksia. Syntaktisesti Selenide-testit ovat ytimekkäämpiä kuin Selenium-testit ja Selenide tekee automaattisesti monia toimenpiteitä, jotka tulee Seleniumissa tehdä manuaalisesti, esimerkiksi selaimen sulkeminen.

4.4 Vertailukriteerien yhteenveto

Tarkastellut testauskehykset voidaan jakaa karkeasti kolmeen ryhmään, yksikkötestaus-, mockaus- ja UI-testauskehykset. Spock Framework poikkeaa hieman tästä jaosta yhdistämällä yksikkötestausta ja mockausta. Testauskehysten toteuttamat ominaisuudet vertailukriteerien suhteen on esitetty taulukossa 1.

Kaikki tarkastellut testauskehykset ovat avoimen lähdekoodin projekteja, mutta ne käyttävät hieman eri lisenssejä. Kaikki käytetyt lisenssit ovat yleisiä avoimen lähdekoodin lisenssejä, jotka mahdollistavat kehysten kaupallisen hyödyntämisen.

5 Pohdinta

Tässä luvussa käymme läpi testauskehysten soveltuvuuden kohdeyrityksen ohjelmistokehitysprosessiin, vastaamme saatujen tulosten perusteella tut-

	Tarjotut testaus-tyypit	Sijoittuminen ohjelmistokehitysprosessiin	Testauksen automatisointi	Suoritettavien testien valinta	Maven-integraatio	Lisenssi
JUnit	Yksikkötestaus	Toteutus ja Testaus	IDE- ja Maven-liitännäinen	Testiluokkien valinta include- ja exclude-säännöillä	Maven-liitännäinen test-vaiheessa	Eclipse Public License 2.0
TestNG	Yksikkötestaus	Toteutus ja Testaus	IDE- ja Maven-liitännäinen	Määritellään testng.xml-tiedostossa	Maven tavoitteet: testng ja testng:junit-report	Apache 2.0
Mockito	Mockaus	Ei kiinteää vaihetta	Ei omaa toteutusta	Ei omaa toteutusta	Riippuvuuksien jakelu	MIT License
EasyMock	Mockaus	Ei kiinteää vaihetta	Ei omaa toteutusta	Ei omaa toteutusta	Riippuvuuksien jakelu	Apache 2.0
Spock	Yksikkötestaus ja Mockaus	Määrittely ja Suunnittelu	Maven-liitännäinen	Maven-Surefire-liitännäisellä	Maven-liitännäinen test-vaiheessa	Apache 2.0
Selenium	Web-selaimen UI-testaus	Testaus ja Hyväksyntä	Ei omaa toteutusta	Ei omaa toteutusta	Riippuvuuksien jakelu	Apache 2.0
Selenide	Web-selaimen UI-testaus	Testaus ja Hyväksyntä	Ei omaa toteutusta	Ei omaa toteutusta	Riippuvuuksien jakelu	MIT License

Taulukko 1: Tarkastellut testauskehykset ja vertailukriteerit

kimuskysymyksiin, arvioimme tutkimuksen luotettavuutta ja vertaamme tutkimusta aiempaan kirjallisuuteen.

5.1 Soveltuvuus kohdeyrityksen ohjelmistokehitysprosessiin

Tarkastelemme testauskehyksiä luvussa 3 esiteltyjen käyttötapauksen kriteerien pohjalta.

5.1.1 Yhteensopivuus Maven-ympäristön kanssa

Kaikkien tarkasteltujen kehysten riippuvuudet voidaan ladata Mavenin avulla. Maven projektissa tämä tarkoittaa, että mitään testauskehystä varten ei tarvitse erikseen ladata tai asentaa mitään ohjelmia tai kirjastoja, vaan

kaikki tarvittavat kirjastot ja komponentit ladataan ja asennetaan projektiin automaattisesti käännöstyökalun toimesta.

JUnit, TestNG ja Spock Framework, eli kaikki tarkastellut yksikkötestauskehykset tarjoavat Maven-liitännäisen, jonka avulla testauskehyksen testit voidaan suorittaa käännöksen yhteydessä. Mockauskehykset, Mockito ja EasyMock, sekä käyttöliittymätestauskehykset, Selenium ja Selenide eivät sisällä omaa testien suoritinta, joten niille ei ole tarjolla Maven-liitännäistä. Mockaus liittyy olennaisesti johonkin toiseen testiin, joten niiden kohdalla on luonnollista, että ne suoritetaan jollakin toisella testien suorittimella. Käyttöliittymätestauksessa testien automatisointi täytyy toteuttaa erikseen, joko erikseen suoritettavalla skriptillä tai ohjelmalla tai jonkin toisen testauskehyksen testien suorittimella manuaalisesti tai käännöksen yhteydessä.

Kohdeyrityksen ohjelmistokehityksessä käytetään Maven-projekteja, joten kaikki tarkastellut kehykset voidaan liittää prosessiin ilman suuria muutoksia.

5.1.2 Testaustasojen saatavuus

Yksikkötestauskehykset tarjoavat suoraan toiminnallisuuden yksikkötesteihin, näitä kehyksiä voidaan lisäksi käyttää integraatio- ja regressiotestaukseen. Integraatiotestit voidaan toteuttaa testin rakenteen kannalta hyvin samalla tavalla kuin yksikkötestit. Erona näiden välillä on testattavat objektit, yksikkötesteissä testataan vain yhtä komponenttia ja integraatiotesteissä testataan useamman komponentin yhteistoimintaa tai jotakin rajapintaa. Regressiotesteissä toistetaan jo toteutettujen komponenttien testejä, jotta voidaan varmistaa, että komponenttien toiminnassa ei ole tapahtunut muutoksia. Regressiotestien kohdalla kysymys on oleellisten testitapausten valinnasta.

Mockauskehysten hyödyllisyys korostuu yksikkötesteissä, vaikka niitä voidaan käyttää millä tahansa testitasolla. Yksikkötesteissä halutaan keskittyä testattavan komponentin toimintaan ja mock-olioita käyttämällä voidaan ohittaa muista komponenteista johtuvien virheiden vaikutus testien tuloksiin. Itsessään mockauskehykset eivät tarjoa testaustoiminnallisuutta, mutta niillä voidaan helpottaa yksikkötestausta. Mockauskehykset ovat erityisen hyödyllisiä testattaessa toiminnallisuutta, joka hyödyntää kolmansien osapuolien hallinnoimia rajapintoja. Esimerkiksi maksullisten rajapintojen tapauksessa on hyödyllistä suorittaa testit mock-oliolla oikean rajapinnan sijasta.

UI-testauskehykset keskittyvät järjestelmä- ja regressiotestaukseen. Vaikka käyttöliittymää voidaan ajatella yksittäisenä komponenttina, sen toimintaa liittyy johonkin suurempaan järjestelmään. Käyttöliittymä ei itsessään tarjoa toiminnallisuutta vaan se on käyttäjän rajapinta ohjelmiston toiminnallisuuteen. On oleellista, että ohjelmiston kehittyessä ja muuttuessa käyttöliittymän toiminnot toimivat määritysten ja käyttäjän odotusten mukaisesti. Selenium ja Selenide tarjoavat tämän testauksen simuloimalla käyttäjän operaatioita selaimessa. Kumpikaan kehyksistä ei tarjoa mahdollisuutta Swing-kehyksellä toteutettujen työpöytäsovellusten testaamiseen.

5.1.3 Käytettävyyden helppous

JUnitin käyttö on yksinkertaista, annotaatiot ja metodit on nimetty selkeästi. `Assert`-metodeita löytyy moniin erilaisiin tarkoituksiin, jolloin niitä voidaan käyttää yksinkertaisella syötteellä ja niille ei tarvitse muodostaa monimutkaisia kutsuparametreja. Testien virheviestit ovat selkeitä ja niissä kerrotaan selkeästi missä kohdassa testiä virhe tai epäonnistuminen tapahtui. Puutteena virheviesteissä on, että niissä ei kerrota mahdollista parametria virheellisessä testissä.

TestNG:n käyttö on hyvin samanlaista kuin JUnitissa. Asioiden nimeäminen ja eri annotaatioiden käyttö ei ole ihan yhtä selkeää, virheviestit ovat hieman epäselvempiä ja esimerkiksi virheellisten tietolähteiden kohdalla TestNG yksinkertaisesti ohitti testin ilman minkäänlaista virheviestiä. Jos käytetään Javan `assert`-avainsanaa epäonnistuneista testeistä ei suoraan anneta selkeää informaatiota, ainoastaan mikä testi epäonnistui ja millä rivillä. Käytettäessä JUnitin `assert`-metodeja epäonnistuneiden testien viestit vastaavat JUnitin viestejä.

Spock Frameworkin käyttö on selkeää, kunhan Groovy-kielen syntaksiin pääse kiinni. Spock Framework-testit ovat ytimekkäitä ja niiden syntaksi on selkeää. Spock antaa selkeitä virheviestejä epäonnistuneista testeistä, joissa näkyy odotetut ja todelliset arvot sekä konteksti, jossa arvoja on käytetty. Mock-olioiden kanssa tulee kiinnittää huomiota siihen, että mockattaessa luokkia rajapintojen sijaan mock-oliot eivät aina täytä Javan luokkaehtoja. Lisäksi huomiota tulee kiinnittää metoditynkien kohdalla siihen, missä tarkalleen ottaen tyngät tulee määritellä, jotta ne toimivat yhdessä mock-olioiden kanssa.

Mockiton käyttö on hyvin yksinkertaista. Testiluokkaan pitää lisätä ainoastaan viite Mockitoon, jonka jälkeen mock-olioiden luominen ja käyttö toimii kuten tavallisten olioiden. Ainoana erona on, että testejä kirjoittaessa tulee muistaa, että mock-olio ei päivitä tilaansa, joten kaikki mock-oliolta saatavat arvot tulee toteuttaa oliolle tynkinä.

EasyMock vaatii Mockitoa enemmän määrittelyjä. EasyMock vaatii JUnit 5:n ja TestNG:n kanssa erillisen laajennusluokan käyttöä, tätä ei vaadita käytettäessä JUnit 4. Lisäksi EasyMock vaatii metodien kutsukertojen määrittelyn, vaikka sitä ei erikseen verifioitaisi. Tämä tuottaa lisätyötä, jos vain halutaan luoda tynkämetodeja mock-oliolle.

Seleniumissa toiminnot määritellään tavalla, joka on web-ohjelmoinnissa tuttua. Sivulta haetaan elementtejä, joille suoritetaan toimenpiteitä. Lopuksi sivulta haetaan jokin tieto, jonka pohjalta testi suoritetaan. Seleniumin suorittaessa testejä se avaa selainikkunan, jota se manipuloi. Suoritus on nopeaa, mutta testien tarkastelu niiden tapahtumisen aikana on mahdollista.

Seleniden testit ovat huomattavasti ytimekkäämpiä kuin Seleniumissa. Seleniden metodien nimet ovat selkeitä ja kuvaavia. Seleniumin tapaan Selenide suorittaa testit selainikkunassa, jota se hallinnoi.

5.1.4 Käyttöönotto

JUnitin käyttöönotto on Maven-projektissa yksinkertaista. Projektin pom.xml-tiedostoon lisätään `<plugin>` ja `<dependency>` osiin esimerkissä 11 olevat tiedot. Tämän jälkeen Maven hakee automaattisesti tarvittavat kirjastot ohjelmiston käännöksen yhteydessä ja testien kirjoittaminen voidaan aloittaa.

```
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.6.2</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.6.2</version>
        <scope>test</scope>
    </dependency>
</dependencies>

<plugins>
    <plugin>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.22.2</version>
    </plugin>
    <plugin>
        <artifactId>maven-failsafe-plugin</artifactId>
        <version>2.22.2</version>
    </plugin>
</plugins>
```

Esimerkki 11: JUnitin pom.xml konfiguraatio.

TestNG:n käyttöönotto Maven-projektissa on hyvin samanlainen kuin JUnitin käyttöönotto. Pom.xml-tiedostoon lisätään esimerkin 12 tiedot. Erona on, että jos TestNG-testisarjoja halutaan konfiguroida tarkemmin, tulee testng.xml-tiedosto luoda erikseen. Jos halutaan suorittaa kaikki testit, TestNG voidaan suorittaa maven-surefire-plugin liitännäisellä ilman testng.xml-tiedostoa samaan tapaan kuin JUnit. Etuna JUnit:iin verrattuna TestNG:ssä on, että kaikki kirjastot saadaan yhdestä riippuvuudesta. TestNG:lle voidaan määritellä erilaisia raportointityökaluja `<property>` tagissa ja TestNG:n mukana tarjotaan työkalu XML-raporttien luomiseen.

```
<dependencies>
  <dependency>
    <groupId>org.testng</groupId>
    <artifactId>testng</artifactId>
    <version>6.9.8</version>
    <scope>test</scope>
```

```
  </dependency>
</dependencies>

<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>3.0.0-M4</version>
    <configuration>
      <suiteXmlFiles>
        <suiteXmlFile>testng.xml</suiteXmlFile>
      </suiteXmlFiles>
      <properties>
        <property>
          <name>reporter</name>
          <value>org.testng.reporters.XMLReporter</value>
        </property>
      </properties>
    </configuration>
  </plugin>
</plugins>
```

Esimerkki 12: TestNG:n pom.xml konfiguraatio.

Spock Frameworkin käyttöönotto tapahtuu Maven-projektissa lisäämällä pom.xml-tiedostoon esimerkissä 13 esitetyt riippuvuudet ja esimerkissä 14 esitetyt liitännäiset. Spock Framework vaatii itse testikirjastojen lisäksi Groovy-riippuvuuksien määrittelyn. Spock Frameworkkia varten tulee määritellä kaksi liitännäistä: maven-surefire-plugin, joka vastaa testien suorittamisesta ja gmavenplus-plugin, joka vastaa Groovy koodin kääntämisestä.

```
<dependencies>
  <dependency>
    <groupId>org.spockframework</groupId>
    <artifactId>spock-core</artifactId>
    <scope>test</scope>
    <version>2.0-M2-groovy-3.0</version>
  </dependency>
  <dependency>
    <groupId>org.codehaus.groovy</groupId>
    <artifactId>groovy</artifactId>
    <version>3.0.2</version>
  </dependency>
  <dependency>
    <groupId>org.spockframework</groupId>
    <artifactId>spock-junit4</artifactId>
    <scope>test</scope>
    <version>2.0-M2-groovy-3.0</version>
  </dependency>
</dependencies>
```

Esimerkki 13: Spock Framework riippuvuudet pom.xml-tiedostossa.

```
<plugins>
  <plugin>
    <groupId>org.codehaus.gmavenplus</groupId>
    <artifactId>gmavenplus-plugin</artifactId>
    <version>1.6</version>
    <executions>
      <execution>
        <goals>
          <goal>compile</goal>
          <goal>compileTests</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
  <plugin>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>3.0.0-M4</version>
    <configuration>
      <useFile>false</useFile>
      <includes>
        <include>**/*Test.java</include>
        <include>**/*Spec.java</include>
      </includes>
    </configuration>
  </plugin>
</plugins>
```

Esimerkki 14: Spock Framework liitännäiset pom.xml-tiedostossa.

Spock Frameworkin mockaus-toiminnallisuus toimii rajapinnoille suoraan. Jos halutaan luoda mock-olioita suoraan luokista, tarvitaan esimerkin 15 mukainen riippuvuus muiden Spock-riippuvuuksien lisäksi.

```
<dependencies>
  <dependency>
    <groupId>net.bytebuddy</groupId>
    <artifactId>byte-buddy</artifactId>
    <version>1.9.3</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Esimerkki 15: Spock Frameworkin mockaus-riippuvuus luokille rajapintojen lisäksi.

Mockiton käyttöönotto vaatii ainoastaan Maven-projektin pom.xml-tiedostoon esimerkin 16 mukaisen riippuvuuden lisäämisen. Tämän jälkeen Maven lataa ohjelmiston käännöksen yhteydessä automaattisesti kaikki riippuvuudet. Mockito ei itsessään toteuta testien suoritusta, joten sen lisäksi tarvitaan jokin testien suoritin, joka vaatii oman käyttöönottonsa.

```
<dependencies>
  <dependency>
    <groupId>org.mockito</groupId>
```

```
    <artifactId>mockito-core</artifactId>
    <version>3.3.3</version>
  </dependency>
</dependencies>
```

Esimerkki 16: Mockiton riippuvuuden määrittely pom.xml-tiedostossa.

EasyMockin käyttöönotossa tarvittavat kirjastot saadaan ladattua esimerkin 17 mukaisella Maven riippuvuudella. Kuten Mockito EasyMock ei toteuta testien suoritusta, joten sen lisäksi vaaditaan jokin testien suoritin. Testatuilla testien suorittimilla, JUnit 5 ja TestNG, EasyMockin käyttöönotto vaati lisäksi laajennusluokan luomisen. EasyMockin oletustoiminnallisuus käyttää JUnit 4 sääntöä, jota ei ole saatavilla JUnit 5:ssä ja TestNG:ssä.

```
<dependencies>
  <dependency>
    <groupId>org.easymock</groupId>
    <artifactId>easymock</artifactId>
    <version>4.2</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Esimerkki 17: EasyMock riippuvuuden määrittely pom.xml-tiedostossa.

Seleniumin käyttöönotossa tulee ladata Seleniumin kirjastot, jotka ladataan lisäämällä Maven-projektin pom.xml-tiedostoon esimerkin 18 mukainen riippuvuus. Seleniumin kirjastojen lisäksi tulee ladata WebDriver-ohjelma selaimelle, jolla testit halutaan suorittaa. WebDriveria valitessa tulee kiinnittää huomiota siihen, että WebDriverin ja asennetun selaimen versiot ovat yhteensopivat. WebDriverin sijainti tulee asettaa PATH:iin, mutta tämä voidaan tehdä testiajon alussa.

```
<dependencies>
  <dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>3.141.59</version>
  </dependency>
</dependencies>
```

Esimerkki 18: Seleniumin riippuvuuden määrittely pom.xml-tiedostossa.

Seleniden käyttöönotto vaatii ainoastaan Selenide riippuvuuden lisäämisen Mavenin pom.xml-tiedostoon esimerkin 19 mukaisesti. Lisäksi Selenide vaatii jonkin testien suorittimen asentamisen, esimerkiksi JUnit tai TestNG.

```
<dependencies>
  <dependency>
    <groupId>com.codeborne</groupId>
    <artifactId>selenide</artifactId>
    <version>5.10.0</version>
    <scope>test</scope>
  </dependency>
```

```
</dependencies>
```

Esimerkki 19: Selenide riippuvuuden määrittely pom.xml-tiedostossa.

5.1.5 Testidatan syöttäminen

JUnit-testeihin testidataa voidaan syöttää käyttämällä `@ParameterizedTest`-annotaatiota. Itse testidata annetaan testille jollakin lähde-annotaatiolla. Parametrillisellä testillä tulee olla parametrit, jotka vastaavat määrältään ja tyypeiltään testidataa. Parametrillisia testejä varten tulee projektin pom.xml-tiedostoon määritellä riippuvuus org.junit.jupiter.params-pakkaukselle, riippuvuus on määritelty esimerkissä 20. Pakkaus sisältää kirjastot parametrillisille testeille ja eri lähdetyypeille.

```
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-params</artifactId>
        <version>5.6.2</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Esimerkki 20: Parametrillisten testien pom.xml konfiguraatio.

Tarjolla olevia lähdetyyppejä ovat arvolähde, tyhjä lähde, enumeraatiolähde, metodilähde, csv-lähde ja csv-tiedostolähde. Samalle testimetodille voidaan määritellä samanaikaisesti useampi lähde, jolloin kaikki määritellyt lähteet käydään peräkkäin läpi. Esimerkiksi arvolähteen ja tyhjän lähteen yhdistämällä voidaan suorittaa testi listatuilla arvoilla ja tyhjällä arvolla.

Arvolähde määritellään `@ValueSource`-annotaatiolla. Arvolähteessä annetaan taulukko, jonka kaikille alkioille testi suoritetaan.

Tyhjä lähde määritellään `@NullSource`- ja `@EmptySource`-annotaatiolla. `@NullSource` antaa testille argumenttina `null` arvon, jolla voidaan suorittaa testi parametrilla, joka ei saa arvoa. `@EmptySource`:a voidaan käyttää parametreille, jotka ovat kokoelmatyyppisiä, kuten `List`, `Set` tai `Map`, merkkijonoja tai taulukoita. `@EmptySource` antaa parametrin, joka kuvastaa tyhjää kokoelmaa tai taulukkoa.

Enumeraatiolähde määritellään `@EnumSource`-annotaatiolla. Enumeraatiolähteellä testi suoritetaan enumeraation kaikille alkioille. Jos esimerkiksi halutaan suorittaa testi kaikilla viikonpäivillä voidaan määrittää enumeraatiolähde `DayOfWeek`-enumeraatiolle.

Metodilähde määritellään `@MethodSource`-annotaatiolla. Metodilähteessä annetaan metodi, jossa testille annettavat parametrit luodaan, metodin tulee palauttaa `Stream`-olio, josta parametrit luetaan testille. Metodilähteellä voidaan toteuttaa monimutkaisempi parametrien luonti.

Csv- ja csv-tiedostolähteet määritellään `@CsvSource`- ja `@CsvFileSource`-annotaatiolla. Csv-lähteissä annetaan merkkijono, joka sisältää csv-muodossa testille annettavat parametrit tai csv-tiedosto. Csv-lähteestä voidaan testille antaa useampi parametri kerrallaan, mutta tällöin testin parametrien tulee vastata määrältä ja tyypiltään csv-datan sisältöä.

TestNG-testeihin testidatan syöttäminen tapahtuu `@DataProvider`-annotaatiolla merkityillä metodeilla. `@DataProvider`-annotaatiolle annetaan parametri name, jolla tietolähde myöhemmin tunnistetaan. Metodien tulee palauttaa joko kaksiulotteinen taulukko, joka on `Object`-tyyppiä, tai iteraattori, joka on `Iterator<Object[]>`-tyyppiä. Taulukon rivit vastaavat suoritettavia testejä ja sarakkeissa on testin parametreja vastaavat arvot. Suoritettavan testin `@Test`-annotaatiolle annetaan parametri `dataProvider`, jolle annetaan arvoksi käytettävän tietolähteen nimi.

Jos testidata halutaan lukea tiedostosta, voidaan tiedoston lukeminen toteuttaa tietolähde-metodissa. TestNG:n testidatan määrittäminen tapahtuu kaikissa tilanteissa samalla tavalla, vaikka se onkin yksinkertaisen testidatan tapauksessa hieman monimutkaista.

Testidataa voidaan syöttää Spock Framework-testeille `where` lohkoissa. Data voidaan joko syöttää manuaalisesti taulukkona testissä tai se voidaan lukea esimerkiksi tietokannasta kuten esimerkissä 21.

```
@Shared sql = Sql.newInstance("jdbc:h2:mem:", "org.h2.Driver")

def "maximum␣of␣two␣numbers"() {
  expect:
  Math.max(a, b) == c

  where:
  [a, b, c] << sql.rows("select␣a,␣b,␣c␣from␣maxdata")
}
```

Esimerkki 21: Spock Framework testidatan lukeminen tietokannasta lähde [14].

Mockauskehyksissä ei suoranaisesti käytetä testidataa. Mock-olioita käytettäessä on tarpeellista huolehtia, että sille luodaan oikeat metodityngät, jotta mock-olion antamat arvot vastaavat testidataa.

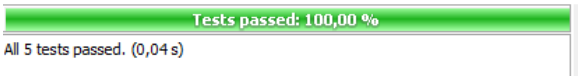
Käyttöliittymätestauskehyksissä ei ole omaa toteutusta testidatan hallinnoinnille. Ajettaessa testiä esimerkiksi JUnitilla voidaan testidata antaa testille JUnitin työkaluilla.

5.1.6 Raportointi

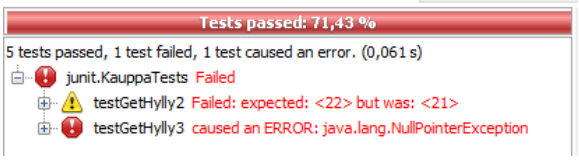
JUnit-testien suorituksesta annetaan raportti ohjelmiston käännöksen yhteydessä. Raportissa eritellään epäonnistuneet testit ja testit, joiden suorituksessa tapahtui virheitä. Jos testit epäonnistuvat tai niiden suorituksessa

tapahtuu virheitä, käännös epäonnistuu.

Suoritettaessa NetBeans-kehitysympäristöstä JUnit testien tulokset näytetään testitulokset (Test Results) ikkunassa. Ikkunassa näkyy vihreä palkki, jos kaikki testit onnistuvat ja punainen palkki jos testejä epäonnistuu. Testien epäonnistuessa ikkunassa annetaan tarkempia tietoja siitä, mitkä testit epäonnistuivat ja mikä oli epäonnistumisen syy. Kuvassa 4 on tilanne, jossa kaikki testit onnistuvat ja kuvassa 5 on tilanne, jossa yksi testi epäonnistuu ja yksi testi aiheuttaa virheen sekä virheilmoitus epäonnistumisista.



Kuva 4: NetBeans-ympäristön raportti onnistuneista JUnit testeistä.



Kuva 5: NetBeans-ympäristön raportti epäonnistuneista JUnit testeistä.

Raportit tulostetaan myös käännöksen konsolisyötteeseen. Kuvassa 6 on raportti onnistuneista testeistä ja kuvassa 7 on raportti epäonnistuneista ja virheellisistä testeistä. Raportin jälkeen seuraa tieto käännöksen onnistumisesta.

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running junit.KauppaTests
Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.039 s - in junit.KauppaTests

Results:

Tests run: 5, Failures: 0, Errors: 0, Skipped: 0

------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time: 4.342 s
Finished at: 2020-04-16T12:43:36+03:00
Final Memory: 20M/309M
------------------------------------------------------------------------
```

Kuva 6: Maven konsolin raportti onnistuneista JUnit testeistä.

Puutteena JUnit-testien raportoinnissa on, että parametrillisen testin epäonnistuessa raportti ei kerro parametria, vaan ainoastaan parametrin indeksin.

TestNG-testeistä saadaan maven-surefire-plugin liitännäisen kautta samanlaisia raportteja kuin JUnit-testeistä. Maven-konsolissa ja NetBeans-ympäristössä näkyvät raportit eivät ole yhtä selkeät kuin JUnitissa, erityi-

Kuva 7: Maven konsolin raportti epäonnistuneista JUnit testeistä.

sesti jos käytetään Javan `assert`-avainsanaa. Käytettäessä JUnitin `assert`-metodeja virheviestit ovat vastaavia kuin JUnitissa. TestNG:n XML-raportit tarjoavat huomattavan määrän informaatiota suoritetuista testeistä. Tähän kuuluu parametrit, joilla testit suoritettiin, testien status ja mahdolliset virheviestit sekä kattava listaus Javan pinosta virheen tapahtuessa. XML-raportointityökalun lisäksi TestNG:lle voidaan luoda muita raportointityökaluja, jotka vastaavat tarkemmin käyttötarvetta ja siten tarjoavat juuri sen tiedon suoritetuista testeistä, joka tarvitaan.

Spock Frameworkin testiraportit ovat saman tyylisiä kuin JUnitissa ja TestNG:ssä. Erona on, että Spock antaa paljon yksityiskohtaisempaa tietoa epäonnistuneista testeistä. Kuvassa 9 ja kuvassa 10 virheviestit, jotka kertovat ehdon, joka ei täyttynyt ja kohdan testikoodissa. Kuvassa 8 nähdään, että Spock Testien nimet ovat luettavampia kuin JUnit ja TestNG testien, koska testit voidaan nimetä ihmisluettavasti metodinimien sijaan. Puutteena raportoinnissa on, että käytettäessä where-lohkoja raportti ei kerro lohkossa annettuja parametreja.

Kuva 8: NetBeans-ympäristön raportti onnistuneista Spock Framework testeistä.

Kuva 9: NetBeans-ympäristön raportti epäonnistuneista Spock Framework testeistä.

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running HyllySpec
Tests run: 3, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.616 s <<< FAILURE! - in HyllySpec
HyllySpec.ota tuotetta hyllystä  Time elapsed: 0.486 s  <<< FAILURE!
org.spockframework.runtime.SpockComparisonFailure:
Condition not satisfied:

hylly.getSaldo() == 0
|     |          |
|     5          false
<fi.vkukkola.example_project.entity.Hylly@32c4e8b2 tuote=fi.vkukkola.example_project.entity.Tuote@4c1f22f3 saldo=5 id=0>

        at HyllySpec.ota tuotetta hyllystä(HyllySpec.groovy:40)


Results:

Failures:
  HyllySpec.ota tuotetta hyllystä:40 Condition not satisfied:

hylly.getSaldo() == 0
|     |          |
|     5          false
<fi.vkukkola.example_project.entity.Hylly@32c4e8b2 tuote=fi.vkukkola.example_project.entity.Tuote@4c1f22f3 saldo=5 id=0>


Tests run: 3, Failures: 1, Errors: 0, Skipped: 0

------------------------------------------------------------------------
BUILD FAILURE
------------------------------------------------------------------------
Total time: 7.199 s
Finished at: 2020-04-16T17:25:43+03:00
Final Memory: 25M/442M
------------------------------------------------------------------------
```

Kuva 10: Maven konsolin raportti epäonnistuneista Spock Framework testeistä.

Mockauskehykset eivät itsessään tarjoa raportointitoiminnallisuutta. Raportoinnin puute voi aiheuttaa ongelmia, jos testiä luotaessa tehdään virheitä mock-olion alustuksen kanssa ja virhe ei johdu testattavasta oliosta vaan mock-oliosta. Erityisen suuri riski tälle on, jos jo olemassa olevia testejä muutetaan käyttämään mock-olioita.

Seleniumin raportit ovat samanlaisia kuin sen testisuorittimen, jossa testit suoritetaan. Seleniumin virheilmoitukset ovat selkeitä. Virheitä voi tapahtua esimerkiksi, jos sivulta yritetään löytää elementtiä, jota ei ole olemassa.

Näissä tapauksissa on oleellista nähdä nopeasti, onko vika testidatassa vai testattavassa sivussa.

Seleniden raportointi on samanlainen kuin Seleniumissa. Lisäksi Selenide ottaa testin epäonnistuessa kuvan selaimesta sekä kopion html-sivusta, jolla virhe tapahtui. Näistä kuvista voidaan tarkastaa sivun visuaalinen tila virhetilanteessa.

5.2 Tutkimuskysymykset

Vastaamme tässä luvussa ja luvussa 4 esitettyjen tulosten pohjalta luvussa 3 esitettyihin tutkimuskysymyksiin. Tutkimuskysymykset ovat:

RQ1. Minkälaisia testauskehyksiä on käytössä ohjelmistonkehitysalalla?

Luvussa 4 on listattu ohjelmistokehitysalalla yleisesti käytössä olevia testauskehyksiä. Testauskehykset voidaan jakaa yksikkötestaus-, mockaus- ja käyttöliittymätestauskehyksiin. Yksikkötestauskehyksiä voidaan hyödyntää yksikkötestauksen lisäksi integraatio- ja regressiotestauksessa. Mockauskehyksillä voidaan eristää yksikkötestien suoritus ainoastaan testattavaan luokkaan ja nopeuttaa testien toteutusta tilanteissa, jossa kaikille testattavan luokan käyttämille olioille ei ole vielä toteutusta. Käyttöliittymätestauskehyksillä voidaan testata web-sovellusten käyttöliittymää ja suorittaa web-sovelluksille järjestelmä- ja integraatiotestejä.

RQ2. Miten erilaiset testauskehykset soveltuvat kohdeyrityksen ohjelmistokehitysprosessiin?

Kohdeyrityksen ohjelmistokehitysprosessissa käytetään Maven-käännöstyökalua Java-ohjelmistojen kehityksessä. Kaikki tarkastellut testauskehykset voidaan liittää helposti Maven-projekteihin. Yksikkötestauskehysten käyttöönotto on kohdeyrityksen ohjelmistokehitysprosessin kannalta oleellista ja ensimmäinen askel testausautomaation liittämisessä prosessiin. Yksikkötestauksen lisäksi tarkastellut yksikkötestauskehykset ovat pohjana muiden testauskehysten suorittamiselle. Mockauskehysten tarjoama testattavien luokkien eriytys auttaa oleellisesti yksikkötestauksen tehostamisessa. Mockauksen tuoma nopeus testauksen aloittamiseen ei ole tarkastellun prosessin kannalta kovin oleellista, koska ohjelmistojen kehityksessä työskentelevä tiimi on niin pieni, että sen hyödyllisyys ei ole merkittävä. UI-testausta voidaan hyödyntää erityisesti web-sovellusten regressiotestaukseen.

RQ3. Mikä on kohdeyrityksen tarpeisiin parhaiten sopiva tapa integroida testaus ohjelmistokehitysprosessiin?

Oleellisin osa testauksen integrointia kohdeyrityksen prosessiin on automatisoidun yksikkötestauksen käyttöönotto. JUnit 5 ja TestNG ovat ominaisuuksiltaan niin samanlaiset, että niiden erot ovat lähinnä mielipidekysymyksiä. JUnit on hieman helpompi ottaa käyttöön, mutta TestNG tarjoaa hieman laajemmat raportointiominaisuudet. Spock Frameworkin etuna on, että testien kirjoittajan ei tarvitse osata Java-ohjelmointia. kohdeyrityksen prosessissa ohjelmoija kirjoittaa testit, joten Java-pohjaisten testien käyttö voi olla haluttavampaa.

Mockauskehysten käyttö ei ole yhtä ehdotonta kuin yksikkötestauksen, mutta se tehostaa yksikkötestien kohdentamista. Mockito on hieman suoraviivaisempi käyttää ja sen yhteensopivuus JUnitin ja TestNG:n uusimpien versioiden kanssa tekee siitä EasyMockia paremman vaihtoehdon.

UI-testauksen käyttöönottoa voidaan pilotoida uusissa web-sovelluksia käyttävissä projekteissa, mutta niiden integrointi vanhempiin projekteihin ei välttämättä ole tarpeellista. Jos uusissa projekteissa UI-testauksen tarjoama hyöty nähdään merkittäväksi, voidaan UI-testauksen käyttöönottoa vanhoihin projekteihin harkita.

5.3 Tutkimuksen luotettavuus

Tutkimuksen tulokset on saatu tarkasteltujen testauskehysten dokumentaation ja testauskehysten käytön perusteella. Testauskehyksille on suoritettu ainoastaan pintapuolinen vertailu, joten vaativimmissa ominaisuuksissa tai pidemmässä käytössä esiin tulevia heikkouksia tai vahvuuksia ei ole huomioitu.

Tutkimusmetodi ei ole kovin systemaattinen, joten on mahdollista, että joitakin testauskehyksiä on tutkimuksessa ohitettu. Tarkastellut testauskehykset on löydetty Google-hakukoneen avulla ei tieteellisistä artikkeleista. Hakukone asettaa oman epäluotettavuutensa, koska Google priorisoi ja valitsee hakutuloksia tuntemattomalla tavalla. Artikkeleissa esitettyjä testikehyksiä on valittu sen perusteella, että niitä on esiintynyt useammassa eri artikkelissa, joten yksittäisen artikkelin kirjoittajan asenteet ja puolueellisuus ei pitäisi näkyä tuloksissa.

Tulokset liittyvät oleellisesti tarkasteltuun Java-versioon ja tarkasteltuun prosessiin. Ohjelmointikielten ja testauskehysten kehittyessä tulokset voivat muuttua oleellisesti. Osa tarkastelluista testauskehyksistä voi siten muuttua enemmän tai vähemmän valideiksi ja uusien testauskehysten julkaisu voi tarjota parempia vaihtoehtoja kuin tässä tutkimuksessa tarkastellut testauskehykset.

Tutkimuksen toistettavuuden isoimpana tekijänä on saatavilla olevien testauskehysten ja ohjelmointikielien kehittyminen. Uusien Java-versioiden

ja testauskehysten uusien versioiden ilmestyessä tutkimuksen tulokset voivat muuttua huomattavasti, mutta tutkimuksen perusperiaatteiden ei pitäisi muuttua merkittävästi. Käytetty Google-hakukone vaikuttaa löydettyihin ei tieteellisiin artikkeleihin, joista testauskehykset valittiin. Google voi tuottaa eri käyttäjille ja eri aikoina samalle haulle erilaisia tuloksia, joten haun toistettavuus ei ole taattavissa.

5.4 Aiempi kirjallisuus

Kirjallisuudesta löytyy useita tutkielman aiheeseen liittyviä artikkeleita. Testauskäytäntöjen ja tekniikoiden vertailusta löytyy useampia tutkimuksia sekä teorian tasolla että alan käytäntöjen vertailussa. Lisäksi löytyy artikkeleja, jotka käsittelevät testauksen osaa ohjelmistokehitysprosessissa. Useat artikkelit keskittyvät kuitenkin joko yhteen testaustyyppiin tai käsittelevät aihetta tätä tutkielmaa rajatummin.

Testauksen liittämistä ohjelmistokehitysprosessiin käsitellään esimerkiksi Pyhäjärven ja Rautiaisen artikkelissa [39]. Artikkeli on kuitenkin lähtökohdiltaan teoreettisempi kuin tämä tutkielma ja siinä ei käsitellä käytännön testauksehyksiä.

Ohjelmistotestauksen käytäntöjä käsitellään alan yrityksissä tehdyssä tutkimuksissa. Nämä tutkimukset usein keskittyvät yhteen testaustyyppiin. Esimerkiksi Engströmin ja Runesonin artikkeli regressiotestauksen tilasta [26] ja Runesonin artikkeli yksikkötestauksen tilasta [41]. Sekä Mostafan ja Yangin tutkimus mockauskehysten käytöstä avoimen lähdekoodin projekteissa [36].

Lisäksi kirjallisuudesta löytyy vanhempia laajoja tutkimuksia testauksen menetelmistä. Esimerkkinä Binderin tutkimus olio-ohjelmoinnin testauksesta [25], joka kokoaa ennen vuotta 1994 julkaistua tutkimusta. Jeffries et al tutkimus erilaisista käyttöliittymäarvioinnin tekniikoista [29]. Sekä Basilin ja Selbyn erilaisia testaustekniikoita vertaileva tutkimus [22], jossa verrataan erilaisia testaustekniikoita ohjelmistoalan ammattilaisten ja opiskelijoiden toimesta.

6 Johtopäätökset

Testaus on olennainen osa ohjelmistokehitysprosessia. Testauksella varmistetaan, että ohjelmisto toimii sille asetettujen määritysten mukaisesti. Manuaalinen testaus on aikaa ja resursseja kuluttavaa toimintaa, jota voidaan testien toistuessa vähentää automaatiotestauksella. Automaatiotestauksessa kirjoitetaan ohjelmallisia testejä, jotka tietokone suorittaa automaattisesti ohjelman käännöksen yhteydessä.

Nykyaikainen ohjelmistokehitys perustuu pääsääntöisesti erilaisiin iteratiivisiin prosesseihin, joissa ohjelmistosta julkaistaan useita versioita sen kehityksen aikana. Ohjelmisto tulee testata jokaisen julkaisun yhteydessä ja on oleellista varmistaa, että sekä vanhat että uudet ominaisuudet testataan

joka julkaisun yhteydessä. Myös ohjelmiston ylläpitovaiheessa sille suoritettavien bugikorjausten ja uusien ominaisuuksien julkaisujen yhteydessä on oleellista testata koko ohjelmisto. Ohjelmiston koon ja julkaisumäärän kasvaessa testauksen määrä kasvaa huomattavasti ja

Mockauksessa luodaan testien yhteydessä mock-olioita. Mock-olio on versio jostakin luokasta, joka ei sisällä mitään toiminnallisuutta, jota sille ei erikseen mockauksen yhteydessä määritellä. Mock-olio pitää kirjaa sille suoritetuista metodikutsuista ja testin yhteydessä voidaan määrittää, mitä kutsuja mock-oliolle pitäisi suorittaa ja testin lopuksi tarkastaa, että vain nämä kutsut ovat tapahtuneet. Mock-olioille voidaan luoda metodityn-kiä, joilla olion metodille voidaan määritellä jokin tietty paluuarvo tietyillä parametreilla.

Mock-olioilla voidaan tarkastaa, että testattavalla luokalla ei ole sivuvaikutuksia. Tarkastamalla mock-oliolle tehdyt metodikutsut tiedetään, että testattava luokka ei aiheuta odottamattomia tapahtumia muissa olioissa. Mock-olioilla voidaan kohdentaa testejä ainoastaan testattavaan luokkaan, jolloin testien epäonnistuessa voidaan olla varmoja, että vika on testattavassa luokassa eikä jossain sen riippuvuudessa. Mock-olioilla voidaan testata luokkia ennen kuin kaikki niiden käyttämät muut luokat on toteutettu korvaamalla vielä toteuttamattomat oliot mock-olioilla. Tällöin luokan testit voidaan kirjoittaa välittömästi riippumatta muun ohjelman tilanteesta. Metodityngillä voidaan varmistaa, että testeissä saadaan halutut arvot. Esimerkiksi virhetilanteiden testaus helpottuu asettamalla metodityngän paluuarvoksi jokin virhe.

Tässä tutkielmassa tarkastelimme ohjelmistokehitysalalla yleisesti käytössä olevia testauskehyksiä Java-ohjelmille. Tarkastelimme yksikkötestaus-, mockaus- ja käyttöliittymätestauskehyksiä. Tarkasteltuja kehyksiä arvioitiin kohdeyrityksen tarpeiden ja ohjelmistokehitysprosessin pohjalta.

Yksikkötestauskehyksistä tarkastelimme JUnittia, TestNG:tä ja Spock Frameworkkia. Nämä testauskehykset tarjoavat toiminnallisuuden yksikkö-, integraatio- ja regressiotestien luomiseen. Lisäksi kehykset sisältävät toiminnallisuuden testien automaattiseen suorittamiseen ja tulosten raportointiin. Kaikki testauskehykset voidaan integroida Maven-käännöstyökalun toimintaan, jolloin ne suoritetaan automaattisesti ohjelmiston käännöksen yhteydessä.

Tarkastellut mockauskehykset ovat Mockito ja EasyMock, lisäksi Spock Framework sisältää oman toteutuksensa mock-olioille. Mockauskehykset sisältävät mock-olioiden ja metodityn kien luomiseen ja käyttöön tarvittavan toiminnallisuuden. Mockito ja EasyMock yhdistyvät luontevasti yksikkötestauskehyksiin ja Spock Framework tarjoaa yksikkötestauksen ja mockauksen samassa paketissa.

Selenium ja Selenide ovat tarkastellut käyttöliittymätestauskehykset. Molemmat testaavat verkkosivujen toimintaa ja niillä voidaan testata käyttäjän toimintaa simuloidussa verkkoselaimessa. Käyttöliittymätestauksella voidaan

varmistaa ohjelmiston toiminta erilaisissa selaimissa.

Kaikki tarkastellut testauskehykset soveltuvat kohdeyrityksen ohjelmistokehitysprosessiin. On oleellista, että prosessiin liitetään automaatiotestaus vähintään yksikkötestauksen tasolla. Automaatiotestaus tulee ottaa käyttöön kaikissa ohjelmistoissa, joiden kehitystä aiotaan jatkaa tai joita ylläpidetään. Mockauksen ja käyttöliittymätestauksen käyttöönottoa voidaan pohtia esimerkiksi jonkin pilottihankkeen muodossa. Tarkastellut kehykset ovat kaikki erittäin kilpailukykyisiä ja kategorioiden sisällä on vaikea asettaa niitä paremmuusjärjestykseen, joten käyttöön voidaan valita mikä tahansa tarkastelluista testauskehyksistä.

Lähteet

[1] *11 top open-source test automation frameworks: How to choose.* `https://techbeacon.com/app-dev-testing/top-11-open-source-testing-automation-frameworks-how-choose`. Accessed: 2020-02-26.

[2] *25 Best Java Testing Frameworks And Tools For Automation Testing (Part 3).* `https://www.softwaretestinghelp.com/java-testing-tools/`. Accessed: 2020-02-26.

[3] *Apache License, Version 2.0.* `https://www.apache.org/licenses/LICENSE-2.0.html`. Accessed: 2020-03-01.

[4] *Best Java Unit Testing Frameworks.* `https://dzone.com/articles/best-java-unit-testing-frameworks`. Accessed: 2020-02-26.

[5] *Cucumber Web Site.* `https://cucumber.io/`. Accessed: 2020-02-05.

[6] *EasyMock Web Site.* `http://easymock.org/`. Accessed: 2020-02-05.

[7] *Eclipse Public License - v 2.0.* `http://www.eclipse.org/legal/epl-v20.html`. Accessed: 2020-03-01.

[8] *JBehave Web Site.* `https://jbehave.org/`. Accessed: 2020-02-05.

[9] *JUnit Web Site.* `https://junit.org/junit5/`. Accessed: 2020-02-05.

[10] *Mockito Framework Site.* `https://site.mockito.org/`. Accessed: 2020-02-05.

[11] *Selenide Web Site.* `https://selenide.org/`. Accessed: 2020-02-05.

[12] *SeleniumHQ Web Site.* `https://selenium.dev/`. Accessed: 2020-02-05.

[13] *Serenity Web Site.* `http://www.thucydides.info/#/`. Accessed: 2020-02-05.

[14] *Spock Framework Web Site*. http://spockframework.org/. Accessed: 2020-02-05.

[15] *TestNG Web Site*. https://testng.org/doc/index.html. Accessed: 2020-02-05.

[16] *Top 10 Java Automation Test Frameworks and Libraries You Can Learn in 2020*. https://dev.to/javinpaul/top-10-java-test-framework-for-automation-in-2019-1mgb. Accessed: 2020-02-26.

[17] *Top 10 Testing Frameworks and Libraries for Java Developers*. https://dzone.com/articles/10-essential-testing-tools-for-java-developers. Accessed: 2020-02-26.

[18] *Top 5 Java Test Frameworks for Automation in 2019*. https://dzone.com/articles/top-5-java-test-frameworks-for-automation-in-2019. Accessed: 2020-02-26.

[19] *What are the Latest Java Test Frameworks for 2019?* https://medium.com/@missshanayaarora/what-are-the-latest-java-test-frameworks-for-2019-243e1383f817. Accessed: 2020-02-26.

[20] *Wikipedia - List of unit testing frameworks*. https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks. Accessed: 2020-02-16.

[21] 24765:2017(E), ISO/IEC/IEEE: *International Standard - Systems and software engineering–Vocabulary*. 2017.

[22] Basili, Victor R ja Selby, Richard W: *Comparing the effectiveness of software testing strategies*. IEEE transactions on software engineering, (12):1278–1296, 1987.

[23] Beck, Kent: *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[24] Beck, Kent, Beedle, Mike, Van Bennekum, Arie, Cockburn, Alistair, Cunningham, Ward, Fowler, Martin, Grenning, James, Highsmith, Jim, Hunt, Andrew, Jeffries, Ron *et al.*: *Manifesto for agile software development*. 2001.

[25] Binder, Robert V: *Testing object-oriented software: a survey*. Software Testing, Verification and Reliability, 6(3-4):125–252, 1996.

[26] Engström, Emelie ja Runeson, Per: *A qualitative survey of regression testing practices.* Teoksessa *International Conference on Product Focused Software Process Improvement*, sivut 3–16. Springer, 2010.

[27] Fewster, Mark ja Graham, Dorothy: *Software test automation.* Addison-Wesley Reading, 1999.

[28] Forsberg, Kevin ja Mooz, Harold: *The relationship of system engineering to the project cycle.* Teoksessa *INCOSE International Symposium*, nide 1, sivut 57–65. Wiley Online Library, 1991.

[29] Jeffries, Robin, Miller, James R, Wharton, Cathleen ja Uyeda, Kathy: *User interface evaluation in the real world: a comparison of four techniques.* Teoksessa *Proceedings of the SIGCHI conference on Human factors in computing systems*, sivut 119–124, 1991.

[30] Leung, Hareton KN ja White, Lee: *Insights into regression testing (software testing).* Teoksessa *Proceedings. Conference on Software Maintenance-1989*, sivut 60–69. IEEE, 1989.

[31] Leung, Hareton KN ja White, Lee: *A study of integration testing and software regression at the integration level.* Teoksessa *Proceedings. Conference on Software Maintenance 1990*, sivut 290–301. IEEE, 1990.

[32] Mackinnon, Tim, Freeman, Steve ja Craig, Philip: *Endo-testing: unit testing with mock objects.* Extreme programming examined, sivut 287–301, 2000.

[33] Memon, Atif M, Pollack, Martha E ja Soffa, Mary Lou: *Hierarchical GUI test case generation using automated planning.* IEEE transactions on software engineering, 27(2):144–155, 2001.

[34] Miller, Roy ja Collins, Christopher T: *Acceptance testing.* Proc. XPUniverse, 238, 2001.

[35] Moniruzzaman, A B M ja Hossain, Syed: *Comparative Study on Agile software development methodologies.* heinäkuu 2013.

[36] Mostafa, Shaikh ja Wang, Xiaoyin: *An empirical study on the usage of mocking frameworks in software testing.* Teoksessa *2014 14th international conference on quality software*, sivut 127–132. IEEE, 2014.

[37] Myers, Glenford J, Sandler, Corey ja Badgett, Tom: *The art of software testing.* John Wiley & Sons, 2011.

[38] Powell, Patricia B: *Software validation, verification, and testing technique and tool reference guide.* Software Validation, Verification, Testing and Documentation, 1986.

[39] Pyhäjärvi, Maaret ja Rautiainen, Kristian: *Integrating testing and implementation into development.* Engineering Management Journal, 16(1):33–39, 2004.

[40] Royce, Winston W: *Managing the development of large software systems: concepts and techniques.* Teoksessa *Proceedings of the 9th international conference on Software Engineering*, sivut 328–338, 1987.

[41] Runeson, Per: *A survey of unit testing practices.* IEEE software, 23(4):22–29, 2006.

[42] Spillner, Andreas, Linz, Tilo ja Schaefer, Hans: *Software testing foundations : a study guide for the certified tester exam : foundation level, ISTQB compliant.* Rocky Nook, Inc, Santa Barbara, CA, 4th edition painos, 2014. `http://login.libproxy.helsinki.fi/login?url=http://www.dawsonera.com/depp/reader/protected/external/AbstractView/S9781492001492`.

[43] Wallace, Dolores R. ja Fujii, Roger U.: *Software verification and validation: an overview.* Ieee Software, 6(3):10–17, 1989.